National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

THE UNIVERSITY OF ALBERTA

LexAGen — A Lexical-Analyzer Generator

by

Randy W.G. Ng

(C)

A thesis

submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree

of Master of Science

Department of Computing Science

Edmonton, Alberta

Fall, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-45533-0

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Randy W.G. Ng

TITLE OF THESIS: LexAGen — A Lexical-Analyzer Generator

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1988

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed)..... Randy W.G. .....

Permanent Address:

22-30 Tai Wong Street East
2/F, Flat B
Wanchai
Hong Kong

Dated July 26, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **LexAGen — A Lexical-Analyzer Generator** submitted by **Randy W.G. Ng** in partial fulfillment of the requirements for the degree of **Master of Science**.

.....

Supervisor

.....

.....

.....

Date: August 2, 1988

for my parents

Abstract

The compilation process consists of source program analysis and code generation. However due to its complexity, source program analysis is further decomposed into lexical analysis, syntax analysis, and semantic analysis. This thesis describes the automatic generation of lexical analyzers. Three major criteria for lexical-analyzer generators are identified: generality, user interface design, and speed. Most current lexical-analyzer generators are deficient in at least one of these aspects. This thesis presents a new lexical-analyzer generator called LexAGen which satisfies all three of these criteria. LexAGen is an integrated programming environment which incrementally creates a lexical analyzer and reports errors immediately as they occur.

Acknowledgement

I thank God for his grace and provisions through all these years of studying overseas.

I would like to thank my supervisor, Dr. Duane Szafron, for his technical and moral support, especially for the long hours of discussion he spent with me that gave this thesis some real substance and many of the ideas in it. I would also like to thank Brian Wilkerson of the Software Productivity Technologies group at the Tektronix, whose inspiration and valuable ideas gave me many insights into this thesis. And I would like to thank Dr. Jim Hoover who made many good comments and suggestions after reading the first draft of this thesis. His immediate feedback is greatly appreciated. Furthermore, I would like to thank Dr. Jonathan Schaeffer and Dr. Werner Joerg for their careful examination of the final draft of this thesis and for their many good comments.

Lastly, but not the least, I would like to express my deepest appreciation to Mei-yuk Leung who through her genuine concern and prayer support, encouraged me all these days.

Table of Contents

Chapter 1. Introduction 1

Chapter 2. Lexical Analysis 5

2.1. Finite Automata 6

2.1.1. Formal Definitions of Finite Automata 7

2.2. Regular Expressions 8

2.3. Equivalence of Regular Expressions and Finite Automata 9

2.4. Approaches to Lexical Analysis Using Finite Automata 11

2.4.1. Interpretation Approach 11

2.4.2. Direct Execution Approach 12

Chapter 3. A Survey of Lexical-Analyzer Generators 14

3.1. Lex — A Lexical Analyzer Generator 14

3.1.1. Design of Lex 15

3.1.2. Lex Source Specification 16

3.2. GLA — A Generator for Lexical Analyzers 18

3.2.1. Design of GLA 19

3.2.2. GLA Source Specification 22

3.3. Mkscan — An Interactive Scanner Generator 22

3.3.1. Design of Mkscan 23

3.3.2. Mkscan Source Specification 24

Chapter 4. Introduction to the LexAGen Environment 26

4.1. Token Specification Using BNF 26

4.1.1. Restricted BNF Notation 27

4.1.2. Using BNF to Specify Tokens 28

4.2. Features of the LexAGen Environment 31

4.2.1. User Interface Paradigm 32

4.2.2. Incrementality 36
4.2.3. Debugging in LexAGen 39
4.2.4. Special Tokens in LexAGen 40
Chapter 5. Implementation of the LexAGen Environment 42
5.1. The Influence of Smalltalk 42
5.1.1. The Smalltalk Environment 42
5.1.2. The Smalltalk Language 43
5.2. Data Representation and Implementation 44
5.2.1. The Browser 44
5.2.1.1. The Left Pane 45
5.2.1.2. The Right Pane 46
5.2.1.3. Other Browser Tables 47
5.2.2. The Specification/Scanner 48
5.2.2.1. Class Automata 48
5.2.2.2. Class AutomataState 49
5.2.2.3. The Construction Process for DFAs 50
5.2.2.3.1. Concatenation 50
5.2.2.3.2. Alternation 51
5.2.3. The LexAGen Strategy for Incremental Analysis 56
5.2.4. The Coder 59
Chapter 6. Keyword Identification 64
6.1. Overview of the Trie-Based Method 65
6.1.1. Special Tries 66
6.1.2. Example of a Pruned O-Trie Forest 67
6.1.3. Searching Time for a Pruned O-Trie Forest 69
6.2. Trie-Index Construction 71
6.2.1. Algorithm for Pruned O-Trie Construction 72

6.2.2. Implementation 76
Chapter 7. A Comparison of LexAGen with Other Scanner Generators 79
7.1. Evaluation of Lex 79
7.2. Evaluation of GLA 81
7.3. Evaluation of Mkscan 82
7.4. Evaluation of LexAGen 84
7.5. Speed Analysis 85
Chapter 8. Conclusion 88
References 90

List of Tables

Table 1.1. Strengths and Deficiencies of Some Lexical-Analyzer Generators 3

Table 6.1. Characteristics for Several Programming Languages 78

Table 7.1. Summary of the Four Scanner Generators 80

Table 7.2. Characteristics of the Input Data 86

List of Figures

Figure 2.1. A Typical Transition Diagram Accepting "ab" or "abc" 7

Figure 2.2. Merging Using Alternation and Concatenation Operations 10

Figure 2.3. Skeleton Program for Interpreting Automata 12

Figure 3.1. General Format of Lex Source 17

Figure 3.2. Lexical Analysis Algorithm for GLA-Scanners 21

Figure 3.3. Excerpts from a GLA Specification for Modula-2 23

Figure 3.4. Implicit State Variable Implementation of Finite Automata 25

Figure 4.1. BNF Specification of Identifiers 27

Figure 4.2. A Typical Automata Browser 33

Figure 4.3. Different Context-Sensitive Menus for the Left Pane 34

Figure 4.4. General and Special Dialog Windows 35

Figure 4.5. Error Reporting Mechanism 38

Figure 4.6. A Typical Deterministic Finite Automaton 40

Figure 5.1. Class Structure of the Deterministic Finite Automata Model 49

Figure 5.2. Concatenation Operation 51

Figure 5.3. Alternation Through Merging 53

Figure 5.4. Merging Involving Non-Terminal 54

Figure 5.5. Ordered Dependency Graph 58

Figure 5.6. Full Expanded Form of a DFA 60

Figure 5.7. Class Structure of the Coder 61

Figure 5.8. Sample C Source Codes Generated 62

Figure 6.1. A Full Trie 67

Figure 6.2. A Pruned Trie 68

Figure 6.3. A Pruned O-Trie 69

Figure 6.4. A Pruned O-Trie Forest 70

Figure 6.5. Implementation of a Pruned O-Trie 77

Chapter 1

Introduction

The compilation process can be decomposed into source program analysis and code generation. However, source program analysis is still a complex task and it can be simplified by further decomposition into lexical analysis, syntax analysis, and semantic analysis. Lexical analysis is the process of identifying the lowest level language constructs (such as identifiers, keywords, labels, and operators). This thesis discusses the automatic generation of lexical analyzers.

Two aspects of lexical analyzers are of concern: the methods of description or specification for the tokens to be recognized, and the methods of recognition itself. There are several specification methods in use, including: regular expressions, pattern matching templates, BNF notations, and simple menus. On the recognition side: automata, continuation tables, and simple branching are used.

Many lexical-analyzer generators exist, especially as parts of compiler writing systems [Lesk75] [Gieg79] [Nurm82]. Perhaps, the best known of them is Lex [Lesk75] which can be used together with the compiler-compiler Yacc [John75] in most UNIX™ systems.

This thesis identifies three major criteria which are important for lexical-analyzer generators. They are:

(1) Generality — In this thesis, generality is a measure of the expressive power

of the grammar of the language which a lexical analyzer can recognize. Most lexical analyzers can recognize languages based on regular grammars. However, many have some restrictions as well as some limited extensions. Therefore, the generality measure is a measure of these restrictions and extensions.

(2) User interface design — There are five aspects that affect ease of use. They are: batch versus interactive mode, incremental development, the amount and form of feedback when errors are made, the amount of guidance and help provided, and the availability of immediate execution to check for specification correctness.

(3) Speed — Lexical analysis is time consuming. A large amount of time is spent reading the source program and partitioning it into the lowest level language constructs. Some measurements have indicated that about 50 per cent of the compilation time is spent tokenising the input text [Wait86a].

This thesis also advocates the view that most current lexical-analyzer generators are deficient in at least one of these aspects. For example, some generators are capable of generating fast, specialised lexical analyzers, while others are capable of generating slow but general ones. Most have poor user interfaces and error reporting facilities. Furthermore, this thesis reviews three lexical-analyzer generators in use today and evaluates them based on these three criteria. They are: Lex [Lesk75], GLA [Wait86b], and Mkscan [Hors87]. Table 1.1 shows the strengths and deficiencies of these generators.

This thesis presents a new lexical-analyzer generator called LexAGen which satisfies all three of these criteria. In fact, these criteria served as the design goals of Lex-

	Generality	User Interface Ease of Use	Speed
Lex	+	-	-
GLA	-	-	+
Mkscan	-	+	+

Table 1.1. Strengths and Deficiencies of Some Lexical-Analyzer Generators

AGen. Although LexAGen has been developed as a stand-alone tool for general use, it is especially well suited for the recognition of common programming languages. LexAGen is not batch-oriented in which a specification is translated into a directly executable lexical analyzer after specification. Instead, LexAGen incorporates the philosophy of integrated programming environments which incrementally create a software product and report errors as they occur.

LexAGen uses its graphical user interface to implement this incremental process. The user specifies a lexical analyzer using BNF productions, and LexAGen incrementally implements this specification as a deterministic finite automaton written in C.

Chapter 2 of this thesis introduces the process of lexical analysis and describes finite automata as a model for the process. Chapter 3 presents the three lexical-analyzer generators which are a representative sample of existing lexical-analyzer generators. Lex represents lexical-analyzer generators which are interpretative. Both GLA and Mkscan represent lexical-analyzer generators which are directly executable. While GLA is batch-oriented, Mkscan uses a full-screen user interface for specification.

Chapter 4 introduces the LexAGen environment from a user's point of view. Chapter 5 focuses on the implementation of LexAGen. Chapter 6 presents a new approach to keyword identification and describes how this approach is implemented in LexAGen. Chapter 7 describes the strengths and deficiencies of the three lexical-analyzer generators in Chapter 3, as well as the strengths and deficiencies of LexAGen. Finally, Chapter 8 concludes with a summary and some suggestions for future enhancements to LexAGen.

Chapter 2

Lexical Analysis

Lexical analysis, or scanning, is the first phase of compilation. Its main task is to transform an unstructured stream of input characters in a source program into fundamental program units, called tokens. For example, ":=" and "<>" are the "assignment token" and "not equal token" in Modula-2, respectively.

Since lexical analysis involves reading the source program from disk, character by character, it is usually considered to be one of the most time-consuming tasks carried out by a compiler [Wait86a]. Therefore, a key design goal of lexical analysis algorithms is minimisation of disk reads and character touches. Specialised buffering techniques for reading input characters and processing tokens can significantly improve the performance of a compiler [Aho86].

Finite automata serve as a good model for the scanning process where each input character represents a transition and each state corresponds to a partial token. Completed tokens are represented by final states. In fact, most generated scanners are implemented as finite automata. This chapter describes the scanning process using finite automata.

Although finite automata are used as a model for scanners and in their implementation, regular expressions are often used to specify scanners. This chapter gives the definition of regular expressions and illustrates how they are equivalent to finite automata. This chapter concludes with a discussion of two implementations of finite automata

which are used in scanners.

2.1. Finite Automata

A finite automaton is a mathematical model that can recognize strings in a language over some alphabet Σ. Informally, a finite automaton consists of a finite number of states and transitions. Each state contains zero or more transitions to other states. Each transition consists of a label and connects exactly two states. At any time, the automaton has one current state. One state of the automaton is labeled as the start state, and it is used as the current state before processing a string. As each character of the string is read, the transition labeled by the input character is used to move from state to state.

Every state is classified as either an *accepting* state or a *non-accepting* state. This classification is used to determine if a string should be accepted as part of the language or not. Consider the situation in which an input character is encountered and no transition labeled with that character exists for the current state, or the situation in which no more input characters exist in the string. In either case, the string is accepted if the current state is an accepting state and rejected if the current state is a non-accepting state.

Finite automata are typically represented by transition diagrams where non-accepting states are denoted by circles, accepting states are denoted by double circles, and transitions are denoted by arcs. Figure 2.1 shows a transition diagram for a finite automaton which represents the language whose valid strings are: "ab" and "abc".

Each finite automaton is either *deterministic*, in that there is at most one transition from a state for each input symbol, or *nondeterministic*, in that more than one transition from a state can be labeled by the same input symbol. Although the same set of

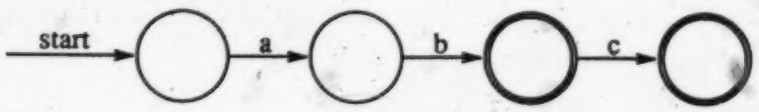


Figure 2.1. A Typical Transition Diagram Accepting "ab" or "abc"

languages can be recognized by both deterministic and nondeterministic finite automata, there is a time-space trade-off between the two. Deterministic finite automata are typically faster than equivalent nondeterministic ones, but they are much bigger.

Sometimes an empty transition between states is useful. That is, the automaton can pass from one state to another without consuming an input character. Such a transition is called an *epsilon* transition.

2.1.1. Formal Definitions of Finite Automata

Definition 2.1. A *finite automaton* (FA) is a 5-tuple (S, Σ, M, S_0, F) where:

[1] S is a finite, non-empty set of states;

[2] Σ is a finite set of input symbols (i.e. the input alphabet);

[3] M is a mapping (depending on the type of FA);

[4] $S_0 \in S$ is the starting state; and

[5] $F \subset S$ is the set of final states.

Definition 2.2. A *deterministic finite automaton* (DFA) is a finite automaton with

$$M: S \times \Sigma \rightarrow S.$$

Definition 2.3. A *nondeterministic finite automaton* (NFA) is a finite automaton with

$$M: S \times \Sigma \rightarrow P(S)$$

where P(S) is the power set of S.

Definition 2.4. A *nondeterministic finite automaton with* ε*-transitions* is a finite automaton with

$$M: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$$

where P(S) is the power set of S.

2.2. Regular Expressions

A regular expression specifies a set of strings in a language over some alphabet Σ. A regular expression contains literal text characters and operators. The text characters are matched with corresponding characters in an input string, whereas the operators are used to specify repetitions and choices. Regular expressions make use of three operations: concatenation, alternation, and closure.

Definition 2.5. Given an alphabet, Σ, a regular expression over Σ can be constructed using the rules:

[1] ε is a regular expression that denotes {ε}, i.e. the set containing the empty string.

[2] If a is a symbol in Σ, then a is a regular expression that denotes $\{a\}$, i.e. the set containing the string a.

[3] Suppose that r_1 and r_2 are regular expressions denoting the

languages $L(r_1)$ and $L(r_2)$ respectively. Then

(a) $(r_1 \mid r_2)$ is a regular expression denoting $L(r_1) \cup L(r_2)$

(b) $(r_1)\,(r_2)$ is a regular expression denoting $L(r_1)\,L(r_2)$

(c) $(r_1)^*$ is a regular expression denoting $(L(r_1))^*$

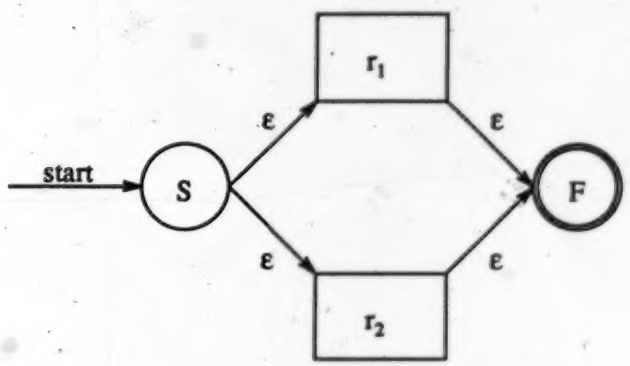
where $L(r_1)\,L(r_2)$ denotes the set of all strings which are concatenations of a string from $L(r_1)$ and a string from $L(r_2)$; and $(r_1)^*$ denotes the set of strings which are concatenations of zero or more strings from $L(r_1)$.

For example, the regular expression, "(a|b)*c", specifies the language consisting of all strings which consist of any number of *a*'s or *b*'s followed by a single *c*. That is, the strings, *bac* and *abaac*, are both in the language along with an infinite number of other strings. It should be noted that regular expressions provide a means of giving a finite specification for a language with an infinite number of strings.

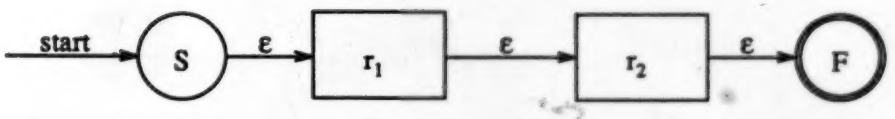
2.3. Equivalence of Regular Expressions and Finite Automata

In general, a regular expression can be converted into a DFA by first using transition diagrams to convert the regular expression into an NFA with ε-transitions and then by converting the NFA into a DFA. The conversion process makes use of a different transition diagram for each of the three operations defined for regular expressions. However, the closure operation has been omitted for brevity.

Consider the operations of alternation and concatenation. Given two regular expressions, r_1 and r_2, and the finite automata which represent them, an NFA with ε-transitions can be constructed when either of the operations is applied. In either case, a new starting



(a) Alternation



(b) Concatenation

Figure 2.2. Merging Using Alternation and Concatenation Operations

state S and a new final state F are created. These new states are not part of the finite automata representing r_1 and r_2.

Alternation is done by inserting ε-transitions from S to r_1 and r_2, and from r_1 and r_2 to F. Concatenation is done by connecting S to r_1, r_1 to r_2, and r_2 to F using ε-transitions. Figure 2.2 shows the transition diagrams for alternation and concatenation.

These equivalences of DFA and regular expressions are established by the following theorems. The proofs of these theorems and methods for construction are given in [Trem85, pp.156-176].

Theorem 1 Given a regular expression **R**, there exists an NFA with ε-transitions, **F**, that accepts the language generated by **R**.

Theorem 2 Let **F** be an NFA with ε-transitions on a language **L**. Then there exists an NFA without ε-transitions, **F′**, such that **L(F) = L(F′)**.

Theorem 3 Let **F** be an NFA without ε-transitions on a language **L**. Then there exists a DFA, **F′**, such that **L(F) = L(F′)**.

Note that the number of states in **F′** blows up exponentially in terms of the number of states in **F**:

$$\text{size } (F') = O(2^{\text{size}(F)}).$$

LexAGen uses DFAs since the speed of a scanner is much more important than its size!

2.4. Approaches to Lexical Analysis Using Finite Automata

A finite automaton provides a simple model for lexical analysis. In fact, lexical analysis is a classical application area of finite automata theory. First, a transition diagram can be easily derived from the lexical specification of a programming language. Secondly, a transition diagram corresponds closely to the syntax graph that describes the structure of the tokens in a programming language.

In general, there are two approaches that can be taken when finite automata are used for lexical analysis. These approaches are called *interpretation* and *direct execution*.

2.4.1. Interpretation Approach

All transitions are grouped together and represented by a single matrix, *table*[*state*, *symbol*], which is interpreted during lexical analysis by a driving program. For example,

```
while (/* basic symbol not complete */)
    switch (state = table[state, nextCharacter]) {
        case state :
            /* code for actions to be performed in each state */
            break;
        ...
    }
```

Figure 2.3. Skeleton Program for Interpreting Automata

if the current state is s_1 and the input symbol is c_1, then the new current state would be the state stored in *table*[s_1, c_1]. An immediate and important observation is that a single algorithm can be used as the basis for interpreting the transition table of all finite automata. The algorithms differ only in the table and the actions that must be taken for each state change.

Thus, this approach allows a lexical-analyzer generator to produce an implementation of the automaton defined by an arbitrary regular language by providing a skeleton interpreter which is augmented by a table and actions for the language specified. Figure 2.3 shows such a skeleton interpreter program.

2.4.2. Direct Execution Approach

A finite automaton can be directly implemented as a high-level language program where the states are represented as different locations in the code and the transition is represented by case statements. There are two major advantages to this approach and one disadvantage.

One advantage is that only the non-error entries in the transition table need to be programmed. The second advantage is that this approach provides superior performance in terms of speed [Heur86] [Wait86a].

The disadvantage is that the directly executable nature of this approach imposes some restrictions on the basic symbol set which can be recognized, if speed is desired. For example, variable names should not start with digits but letters; otherwise, it becomes difficult to distinguish between identifiers and numbers. This means that the implementation of an arbitrary automaton usually requires extensive hand coding.

Fortunately this disadvantage can be addressed. Although direct execution limits the basic symbol set, common programming languages are not arbitrary and their specialised nature reduces the impact of this restriction. For example, identifiers start with letters and follow by zero or more letters or digits.

Chapter 3

A Survey of Lexical-Analyzer Generators

For the sake of efficiency, lexical analyzers for production compilers are often hand-coded. However, lexical-analyzer generators can produce efficient scanners [Aho86] [Möss86]. In addition, generated scanners have certain advantages. Generated scanners can be produced quickly and easily. For example, although scanner generators utilise the best pattern-matching algorithms, the individual who uses one needs to know nothing about pattern matching. Automatically generated scanners also have fewer programming bugs. Furthermore, the lexical description used to produce a generated scanner is not only a specification, but also serves as valuable documentation for the generated lexical analyzer.

In this chapter, several lexical-analyzer generators will be examined closely in terms of their design and methodology of specification which they use. Although many other scanner generators are in use today, the three generators described in this chapter are a representative cross-section. In chapter 7 of this thesis, the performance of scanners generated by these scanner generators will be compared to the performance of scanners generated by LexAGen.

3.1. Lex — A Lexical Analyzer Generator

The best known and most widely used scanner generator is Lex [Lesk75]. Lex is a general-purpose lexical-analyzer generator designed to accept a high-level, problem-

oriented specification for character string matching. It generates a deterministic finite automaton from the regular expressions given in the source, and the generated automaton is interpreted. Lex can be used alone for simple transformations, or for analysis and statistics gathering on the lexical level.

3.1.1. Design of Lex

Lex was designed to generate scanners for general applications. That is, it is not restricted to scanners for programming languages. For example, it is well suited for editor-script type transformations.

The generality of Lex can be seen in the freedom that Lex allows its users to exercise. The Lex specification (source) file is composed of a table of regular expressions (rules) and associated program fragments called actions. Lex generates a scanner which executes an action each time its associated expression accepts a token. A token is usually returned as a value.

Lex can generate a lexical analyzer for an ambiguous specification. When more than one expression can be used to accept the current input string, Lex chooses one of the rules according to the meta-rules:

[1] Use the rule which accepts the token of greatest length.

[2] If two rules can be used to accept a single string as two different tokens of the same length, then use the rule which appears first in the specification.

The second meta-rule can be used to distinguish reserved words from identifiers in a programming language when the reserved words meet the general criteria for identifiers. This is accomplished by ensuring that the rules for the individual reserved words precede

the rules for general identifiers.

Lex supports multiple-character lookahead. However, the input stream will be backed up to where the end of the accepted token is, and all of the lookahead characters must be touched (read) again. Lookahead can be deliberately invoked by the user to perform complex pattern-matching. A good example of this is the DO-statement in FORTRAN. Since blanks are insignificant outside of comments and Hollerith strings in FORTRAN, a DO-statement and an assignment statement starting with DO are difficult to distinguish. For example, after removing insignificant blanks, the DO-statement:

DO100I=1,5

and the ASSIGNMENT statement:

DO100I=1.5

can be differentiated using lookahead. In Lex, the DO-keyword can be specified by:

DO / {digit}+ ({letter} | {digit})* = ({letter} | {digit})+ ,

where the character, /, is the lookahead operator. When the comma is read by Lex, the DO is returned as the keyword token and input is backed up to the character following the character 'O' in DO. Substantial lookahead can be said to be a virtue in Lex since it can find context sensitive tokens.

3.1.2. Lex Source Specification

The general format of Lex source consists of three parts: declarations, rules, and user subroutines [Lesk75, p.8]. This format is shown in Figure 3.1.

The declaration section includes declarations of variables, manifest constants, and regular definitions. A variable declaration is that of a target language variable which is

```
{declarations}
%%
{translation rules}
%%
{auxiliary functions}
```

Figure 3.1. General Format of Lex Source

used in the code for actions, for example:

```
char buffer[1024];
```

where the variable definition must be: (1) preceded either by a space or tab, or (2) enclosed by the delimiters, %{ and %}.

A manifest constant is a constant declaration used in the target language, for example:

```
#define EQ 1
```

where the constant declaration must begin in column 1. Since the above statement is a preprocessor statement, two lines containing the delimiters, %{ and %}, are required to precede and follow the above statement, respectively.

A regular definition is a name which is bound to an expression, so that the name can be used in place of the expression in the rules. This allows the same expression to be used in different rules without repeating it, for example:

```
digit    [0-9]
```

where the regular definition must begin in column 1. When used, the name must be enclosed by the delimiters, { and }, for example, {digit}.

The translation rules represent the user's control decisions. Each transition rule consists of a regular expression and an action. The definitions of regular expressions are similar to those in the previous chapter.

Classes of characters can be specified using the operator pair []. For example, the expression, [*abc*] matches a single character which may be an *a*, *b*, or *c*. The operator, -, indicates a character range. For example, [*a-d*] is equivalent to [*abcd*].

Repetition is specified using the Kleene star, *, and Kleene plus, +, operators. The star means zero or more repetitions and the plus signifies one or more repetitions. For example, [*A-Za-z*][*A-Za-Z0-9*]* represents all alphanumeric strings which start with an alphabetic character, while the expression, [*0-9*]+, specifies the set of all unsigned integers. Notice that concatenation is specified by juxtaposition. In addition to the operators, *, +, [], and -, there are many other operators. The complete set of operators is given in [Lesk75, pp.3-5]:

"\[]^-?.*+|()$/{}%<>

For example, parentheses are used for grouping and the operator, |, is used for alternation. To use any operator as a text character, it must be escaped with the character, \

3.2. GLA — A Generator for Lexical Analyzers

GLA [Wait86b] is a tool designed to build fast, directly executable lexical analyzers from a specification of basic symbols found in the language to be recognized. GLA is not a general-purpose lexical-analyzer generator. Unlike Lex and other general-purpose lexical-analyzer generators which accept arbitrary regular expressions and actions, GLA imposes some restrictions on the structure of the source language to be recognized.

3.2.1. Design of GLA

GLA partitions all basic symbols used in programming languages into three categories: identifiers, denotations, and delimiters. Identifiers are entities whose meanings are determined by the parser. For example, an identifier in a declaration, an identifier in an assignment statement, and a keyword are treated in a similar fashion by the generated scanner.

Denotations represent constant values in the universe of the source language. For example, 3.5, 'Hello World', and TRUE are denotations in Modula-2.

Delimiters are operators or program structuring symbols that carry no information beyond their recognition by the compiler. For example, comment delimiters, string delimiters, parentheses, and white space are delimiters.

A GLA token is not simply a value. It is a record containing other information about the token. Each token also contains its coordinates (line number and character position) in the source program for error reporting, its corresponding terminal code (token value), and an intrinsic attribute if it exists. Identifiers have a symbol table entry as an attribute, and denotations have an entry into a constant table as an attribute.

In addition to the lexical analysis module which contains the algorithm for recognizing basic symbols, GLA makes use of four other supporting modules which provide services to the lexical analysis module. The four supporting modules are: the source text module, the symbol table module, the constant table module, and the error-reporting module.

The source text module represents the source program as a stream of characters and

provides these characters to the scanner as required. This module provides the efficient disk access required by the scanner.

The symbol table module stores the set of distinct identifiers used in the program, including keywords if they are lexically the same as the identifiers. The symbol table is actually pre-loaded with the keywords defined for the source language, each of which is given a proper terminal code and a null intrinsic attribute.

The constant table module maintains the values of the constants used in the source program. For example, literal strings, integers, and floating point numbers can be stored.

The error-reporting module contains the set of error messages appropriate for the source language. These include the error messages generated during scanning as well as those generated during other phases of compilation.

The strategy of the lexical analysis algorithm is to select a subautomaton on the basis of the class of a token's first character. Basic symbols found in most programming languages can be grouped into six classes:

[1] characters which begin identifiers;

[2] characters which begin integer and floating point numbers;

[3] character(s) which begin strings;

[4] character(s) which begin comments;

[5] single-character, non-alphanumeric symbols;

[6] multiple-character, non-alphanumeric symbols.

This selection of subautomaton is controlled by a character-indexed table called

```
do {
    /* skip white space */
    switch (terminal = chtbl[currentCharacter]) {
        case NULT: /* any other characters */
            /* check for line and file terminators, and illegal characters */
            break;
        case IDNT :
            while (scantbl.identifier[currentCharacter])
                advance one character;
            break;
        ...
    }
} while (/* terminal is NULT */);
```

Figure 3.2. Lexical Analysis Algorithm for GLA-Scanners

chtbl, each element of which specifies a terminal code (token value) for some basic symbol beginning with the indexing character. After selecting a subautomaton from one of the first four classes, further automaton actions are determined by another character-indexed context table, *scantbl*. For example, Pascal needs 66 terminal codes and four contexts: scantbl.identifier, scantbl.integer, scantbl.string, and scantbl.comment. Additional contexts may be needed for denotations, for example, floating point numbers, binary numbers, and so on.

This table contains contexts of binary elements which indicate whether the indexing character is a valid continuation for a token of the appropriate type. Figure 3.2 shows some GLA skeleton code for scanning including the use of *chtbl* and *scantbl*.

For the remaining two classes, [5] and [6], further action is determined by the nature of the symbols in these classes.

3.2.2. GLA Source Specification

GLA uses a very high-level specification language. It obtains information about the form and coding of basic symbols from two sources:

(1) Token definitions: A specification of the basic symbols having intrinsic attributes, the desired forms of comments, and special options not otherwise appearing literally in the list of basic symbols as described in (2).

(2) Basic symbol list: A list of all basic quoted and unquoted symbols (to be described later) of the source language, possibly with a terminal code (token value) attached to each.

The token definitions contain one line for each basic symbol with intrinsic attributes, two lines for each desired form of comment, and one line for each option to be exercised. Each line, in turn, consists of a keyword that defines the purpose of the line, plus other elements (maybe integers, strings, or sets) needed to complete the definition.

The basic symbol list consists of two kinds of terminals: quoted and unquoted terminals. A quoted terminal represents a single symbol appearing literally in the source program, while unquoted terminals represent sets of symbols. For example, '+' and 'BEGIN' are quoted basic symbols, while Name and Real_Number are unquoted basic symbols representing variable names and real numbers, respectively.

Figure 3.3 contains excerpts from a GLA specification for Modula-2.

3.3. Mkscan — An Interactive Scanner Generator

Mkscan [Hors87] is a tool for generating and editing scanners. It is not batch-

```
':='.
';'.
...
'BEGIN'.
'END'.
...
Modula_identifier.
Modula_integer.
Modula_real.
Modula_string.
...
```

(a) Basic Symbol List

```
IDENTIFIER Modula_identifier [A-Za-z] [A-Za-z0-9]
INTEGER Modula_integer [0-9] [0-9A-Fa-f] [BCH]
REAL Modula_real [E] [0-9]
STRING Modula_string ['"]
BEGINCOMMENT 1 (*
ENDCOMMENT 1 *)
```

(b) Token Definitions

Figure 3.3. Excerpts from a GLA Specification for Modula-2

oriented like Lex and GLA. Instead, it is an interactive tool with a full screen user interface. Like GLA, Mkscan has been tailored to suit the lexical structure of common programming languages.

3.3.1. Design of Mkscan

Mkscan is easy to use. It requires no prior knowledge of regular languages or other aspects of compiler theory. Instead, it guides the user through a series of choices in an interactive manner.

To accomplish this interaction style, it was necessary to sacrifice generality. In particular, Mkscan assumes that input processing will be performed in a free-format mode. Furthermore, tokens are classified according to their common use in programming languages and grouped into four categories: identifiers, keywords, numbers, and special symbols.

Mkscan makes it easy to modify previously generated scanners. It does this by storing information about the scanner as comments at the beginning of the file which contains the scanner. If a user wishes to modify a scanner, the scanner's file is read by Mkscan and the comments are used as the specification. Notice that when a scanner is generated, the user gets the specification along with the scanner.

Mkscan generates scanners which are compact and fast, with flexible interfaces for external programs. The scanners are implemented as finite automata using a direct execution approach. To be more precise, Mkscan has adopted an implicit state variable implementation of finite automata as shown in Figure 3.4.

3.3.2. Mkscan Source Specification

Mkscan guides the user through a series of menu pages, organised in a hierarchical manner. At the top level, the main menu contains choices for (1) input of a previously generated scanner, (2) editing of an existing (or new) scanner, and (3) output of the scanner.

The process of editing is broken down into several submenus. Each submenu corresponds to a class of lexical tokens or some other main lexical feature of scanning. Mkscan assumes that the input stream will contain identifiers with a certain and regular

```
State_X:
    switch (currentCharacter) {
        case 'a' : goto State_Y;
        ...
    }
    return;
State_Y:
    ...
```

Figure 3.4. Implicit State Variable Implementation of Finite Automata

structure. It has assumed that an identifier consists of one or more characters, where the first character is drawn from one set of characters and the remaining characters from another set of characters. Moreover, keywords are assumed to satisfy the rule for identifiers. That is, a keyword must be a valid identifier in itself. A hashing technique is used to determine if an identifier found is actually a keyword specified by the user.

Chapter 4

Introduction to the LexAGen Environment

LexAGen is an interactive software environment for the design, coding, and testing of lexical analyzers. It is capable of producing fast, general-purpose lexical analyzers. However, LexAGen has also been designed to cope with lexical structures in programming languages. Unlike most general-purpose lexical-analyzer generators, LexAGen does not use regular expressions for specifying lexical analyzers. Instead, it utilises a restricted form of BNF as a conceptual framework for specifying tokens like identifiers, strings, and comments found in common programming languages. The user interactively edits BNF productions to specify a scanner.

This chapter begins by describing the basis for the specification notation used in LexAGen. However, the most important innovation in LexAGen is the extent of assistance it provides to guide the user through the process of design, coding, and testing. This assistance is based on a graphical user interface, and this chapter describes the special features of LexAGen which are based on that interface.

4.1. Token Specification Using BNF

Usually, the lexical structure of a programming language is specified by using a regular grammar, in which tokens are built according to some simple syntactic rules. LexAGen uses the formalism of BNF to define its tokens for three reasons. Firstly, BNF has been used extensively in the formal definition of the syntax of most programming

```
<identifiers>              ::= <letter> | <letter> <alphanumeric characters>
<alphanumeric characters>  ::= <letter>
                           |   <letter> <alphanumeric characters>
                           |   <digit>
                           |   <digit> <alphanumeric characters>
<letter>                   ::= A | ... | Z | a | ... | z
<digit>                    ::= 0 | ... | 9
```

where the ellipses represent a short-hand notation of ranges.

Figure 4.1. BNF Specification of Identifiers

languages, so most users are familiar with it. Secondly, BNF is often used in parser specification; thus, there is no need to introduce a new notational framework when parser generation is integrated with LexAGen. And lastly, the DFA representation used in LexAGen is closer to BNF so that future direct computation of DFAs would be much easier to understand if BNF is used. In fact, LexAGen uses a restricted form of BNF together with some special mechanisms.

4.1.1. Restricted BNF Notation

In BNF notation, there are two kinds of symbols, terminal symbols which are strings of characters, and non-terminal symbols which are symbols that represent the alternation of the concatenation of other terminal and non-terminal symbols. Non-terminal symbols are enclosed in angle brackets, <>, so they can be differentiated from the terminal symbols.

Besides the angle brackets, two other meta-symbols are used. The vertical bar, |, is used to represent alternation and the meta-symbol, ::=, is used to define non-terminals.

Each non-terminal definition is called a production. Figure 4.1 includes four BNF productions which together define identifiers in Pascal or Modula-2.

The use of BNF notation does not overly restrict the expressive power of the lexical-analyzer generator. BNF offers a good notational framework for context-free grammars. Since context-free grammars are a superset of regular grammars, BNF is actually more general than regular expressions. However, in LexAGen, the BNF productions are restricted in that recursive non-terminals can only appear as the rightmost symbol in a production. This means that the context-free grammars are restricted to be right-linear which are equivalent to regular grammars [Reve83]. The restriction of right linearity serves to eliminate cycles in the DFA representation of the specification.

Extended BNF notation is one alternative to BNF notation. Extended BNF includes the Kleene cross, C+, which means a sequence of one or more strings from the syntactic category, C. Another such extension is the Kleene star, C*, which stands for a sequence of zero or more elements of the class, C. These two notations have made the extended BNF notation a little more descriptive; however, one can easily employ the BNF notation to express the same ideas using recursive definitions. That is, the expressive power of BNF is equivalent to the expressive power of the extended BNF.

4.1.2. Using BNF to Specify Tokens

When BNF is used to specify tokens, one additional pair of metacharacters is needed. While some non-terminals represent tokens, others are used to make the definitions more readable or to give names to commonly occurring expressions. The two types of non-terminals must be differentiated since the generated scanner must only

return tokens. The metacharacters, << and >>, can be used to identify those non-terminals which are tokens.

LexAGen has been designed to cope with the lexical structure of programming languages. Unfortunately, the restricted BNF notation is not capable of representing all of the tokens which are commonly found in programming languages. For this reason, additional mechanisms are needed to specify these *special* tokens.

There are basically three kinds of special tokens. The first kind of special token is used to specify identifiers and keywords. An identifier, in itself, is a general token which may be specified using BNF. However when used in conjunction with programming languages, identifiers can be special tokens since keywords often have to satisfy the specification for identifiers and yet must be differentiated as different tokens. LexAGen provides a mechanism for specifying identifiers and keywords.

The second kind of special token is used to specify strings and comments. Sometimes, it is necessary to exclude some characters from the semantic value of a token. A string is such a token, in that the enclosing quotes are not part of the string text. Sometimes, on the other hand, it is necessary to ignore some text when seeking the next token. A comment is such an entity. Although simple comments can be specified using the restricted BNF notation, nested comments cannot. Since some programming languages (like Modula-2) allow nested comments, a special mechanism must be used to specify them. LexAGen allows both strings and nested comments to be specified. The key to recognizing nested comments is to keep track of the nesting level.

The third kind of special token is one in which simple lookahead is required to

resolve ambiguity. For example, the sequence ">=" may be interpreted either as single token or as two tokens, ">" and "=". As is the case with most lexical-analyzer generators, LexAGen resolves this kind of ambiguity by choosing the longest match (i.e. by recognizing the token ">=" in this example). In some other cases, however, this method is not appropriate or adequate. In Modula-2, for instance, the sequence "123.." would be recognized as the real constant "123." and the decimal constructor ".". However, the correct interpretation should be the cardinal constant "123" and the range operator "..". This problem has been addressed and solved in the Alex scanner generator [Möss86].

In LexAGen, the user can force this result by attaching two lookahead characters to the end of an alternative of a production. When the last lookahead character is found, it is left in the input stream (as untouched). In addition, the input stream is rewound one character, and the contents of the token buffer are filled with characters since the last match and up to (but not including) the rewound character. The production's token value is returned. Consider the following production for <cardinal>:

<cardinal> ::= <unsigned integer><..

where the metacharacter, <, has been used to indicate that the next two characters are the lookahead characters. This production can be used to solve the range operator problem of Modula-2. When the first dot is found, the next character is examined in the usual manner. If it is a dot, then the lookahead is successful. Thus, the second dot is left untouched, while the input stream is rewound. The first dot becomes the current character. The contents of the token buffer consist of the string "123", and the token value, CARDINAL, is returned to the client application. The next token found will then be the range operator "..". Notice that this mechanism is really a two-character lookahead since

both dots are examined in capturing the token, "123", while the mechanism of longest match essentially involves one-character lookahead.

LexAGen is currently restricted to this two-character lookahead scheme since two-character lookahead is basically sufficient for most modern common programming constructs. However, it is generally easy to adapt other fixed-length lookahead schemes. Although Lex can solve the Modula-2 range operator problem using its multiple-character lookahead facility, Mkscan and GLA must use special, restrictive built-in mechanisms.

4.2. Features of the LexAGen Environment

To discuss the way in which the BNF based specification is used in LexAGen, it is necessary to look at the environment as a whole. LexAGen is the first component of an envisioned integrated compiler generation environment. As such, it can be viewed as the first component of a specialised programming environment. Integrated programming environments are usually described as those that support software creation, modification, execution, and debugging. One goal of integrated programming environments is to build tools that share a common internal representation of the underlying software structure. A second goal is to present a consistent user interface across tool boundaries. However, there is a third goal which is often missing. An integrated programming environment encapsulates the mechanisms used to implement the environment's functionality. For example, the same syntax checking mechanism can be used throughout the environment.

The graphical user interface of LexAGen is an example of such an encapsulation mechanism. In general, graphical user interfaces can have many positive effects on

programming environments: error reduction, simplified incremental analysis, and efficient debugging [Deli84]. LexAGen is unique among scanner generators in applying these benefits to scanner generation.

4.2.1. User Interface Paradigm

LexAGen is implemented in Smalltalk-80 [Gold85], and the user interface is based on the Smalltalk-80 interface. The user interacts with LexAGen using a special browser, pop-up menus [Gold84], and dialog windows [Szaf86].

The browser is divided into the left and right panes. Each pane represents a different context in which the user can perform specific tasks appropriate for the overall process of developing a lexical analyzer. The names of non-terminals appear in the left pane. At any time, a single production from the left pane is highlighted and the alternatives for that non-terminal appear in the right pane. Different non-terminals can be selected by clicking the mouse on the desired non-terminal in the left pane. Figure 4.2 shows the browser.

Commands are given to LexAGen through the use of pop-up menus. Each browser pane or dialog window has a menu associated with it. The menus are sensitive to both the context (pane or window containing the cursor when the menu pops up) and the current internal state of the context. This allows LexAGen to display those menu operations that are applicable at a given time for a given context. This reduces the number of errors which can be made by the user. For example, Figure 4.3 shows two different pop-up menus for the same production selected in the left pane in which the selected production (1) has not been and (2) has been defined as a token. Note that the second pop-up menu is a hierarchical one.

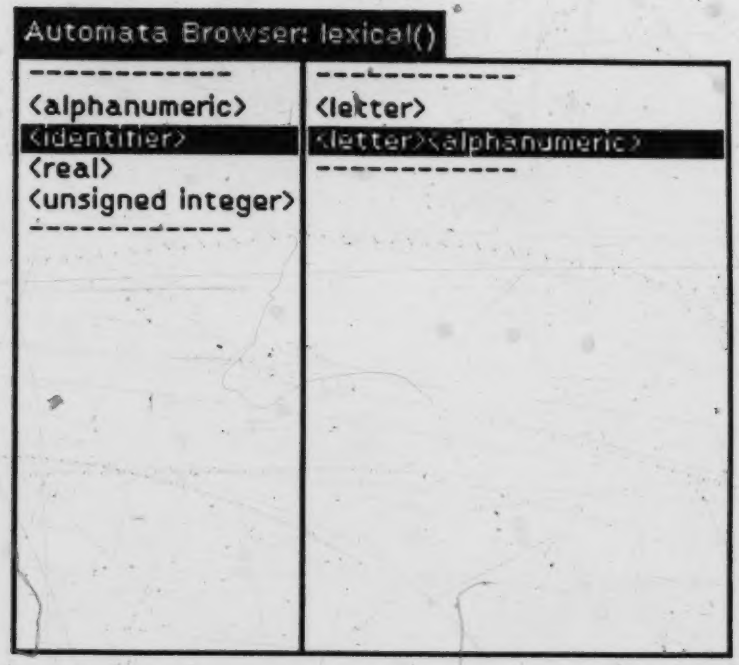
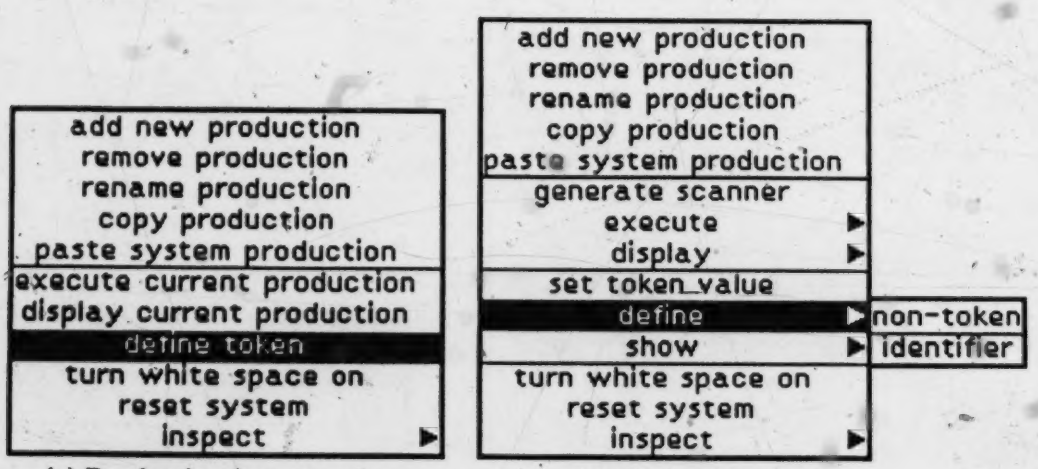


Figure 4.2. A Typical Automata Browser

Each command takes a fixed number of arguments. If a command has several fixed options, then LexAGen presents a hierarchical menu with the appropriate options. If a command requires input from the user, LexAGen then presents a dialog box to the user requesting the necessary input. Figure 4.4(a) shows such a dialog box.

To enter a new production, the user chooses the "add new production" command from the left pane menu. A general dialog box appears, and the user types the name of the non-terminal. When the "accept" command is chosen from the menu for the dialog window (or the return key is pressed), the name appears in the left pane of the browser and is

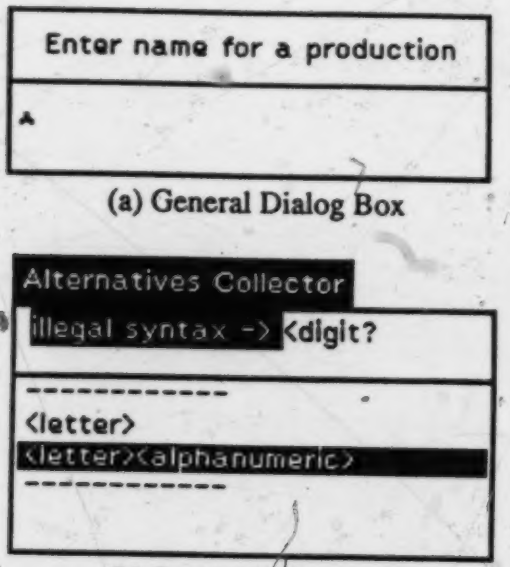


(a) Production is not a token (b) Production is a token

Figure 4.3. Different Context-Sensitive Menus for the Left Pane

highlighted. To enter a BNF expression for this non-terminal, the user chooses the "edit alternatives" command from the right pane menu. A special dialog window, as shown in Figure 4.4(b), appears and the user types the expression in the top text-entry view and chooses the "accept" command from the text menu (or presses return). The expression will appear in the bottom selection view of the dialog window and in the right hand pane of the browser. If other special dialog windows are opened for the current production, they are also updated to reflect this addition of a new expression.

The set of menus responding to the bottom selection view of the special dialog box is exactly the same as the set of menus responding to the right hand pane of the browser. This allows the user to enter multiple expressions for the same production without having



(a) General Dialog Box

(b) Special Dialog Box

Figure 4.4. General and Special Dialog Windows

to choose the "add new alternative" many times (this was the original command available in the right pane instead of the "edit alternatives" command). Figure 4.4(b) shows a special dialog window in which an error message has been inserted in the text view to report a syntax error.

Note also that the graphical user interface obviates the need for the alternation metacharacter, |, and the non-terminal defining meta-symbol, ::=. Although the user must supply the metacharacters, < and >, to differentiate between non-terminals and terminals in the right pane, LexAGen supplies the brackets in the left pane. LexAGen uses an escape metacharacter backslash, \ so that an angle bracket, <, can be entered as a literal. Of

course, to enter a literal backslash, two backslashes must be used. As a design decision, LexAGen generates scanners which assume free-format. That is, white spaces (one or more blanks, tabs, or newline characters) are token separators, so they cannot be escaped. They are simply ignored.

When the name of a non-terminal appears in the left pane, a pair of single angle brackets is supplied automatically by the browser to denote that the non-terminal is non-token by default. However, there is a menu command which the user can use to declare that the non-terminal is really a token. When this command is executed, the single angle brackets are replaced by a pair of double angle brackets to denote that the non-terminal is now a token. Note that the user can also *un*-declare a token at any time by choosing the appropriate menu command.

For each token, the user can also specify a token number to be returned by the final lexical analyzer. If no token number is assigned by the user, then a default value is provided.

4.2.2. Incrementality

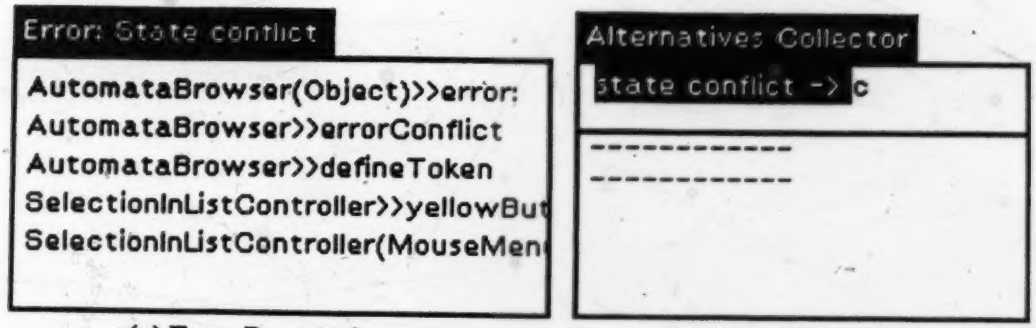
Graphical user interfaces have an immediate impact on the issue of incrementality in programming environments. They can simplify the problems of incremental analysis by requiring the user to enter information in special dialog boxes (entry windows). As correct information is accepted, it can be transferred to other windows which maintain a representation of correct structures. Incorrect information can simply be left in the entry windows until other changes result in its final correctness [Szaf86]. For example, an expression which might result in a conflicting token definition might have to remain in an

entry window until other expression(s) or production(s) is (are) deleted to remove the potential conflicts. This helps the underlying environment not only cope with incorrect information easily, but more importantly a correct representation of the internal structure is always maintained. An additional advantage of this approach is that the user receives immediate feedback as to the correctness of a specification.

LexAGen uses this approach to eliminate both syntax errors and token specification conflicts. Figure 4.5(a) shows an error message which results from an attempt to declare a conflicting production as a token. However, some specific incorrect information is accepted in LexAGen, namely undeclared production names which are used in the right pane of the browser. For example, the production, <binary digit>, may be used in the right hand side of a production, even though the name, "binary digit", has not yet been defined in the left hand pane.

This behavior has been allowed for user convenience. In fact, declared and undeclared references share the same internal representation, but they are kept separately. Their existence does not harm the process of incremental analysis, nor the final recognition of expressions. At any time the user may issue a command which lists all the names which are currently undeclared.

The simplest approach to the problem of incremental analysis is to determine the minimal, separate unit of incrementality. The major goal of incremental analysis is to avoid re-analyzing the entire structure or large portions of it, whenever small changes are made. In other words, the time to update the internal state of the system should be minimal when a small change takes place.



(a) Error Reported (b) Error Inserted

The single underlying internal and intermediate representation of LexAGen is the deterministic finite automaton (DFA). Each production is represented as a DFA, whose structure is the smallest unit of incrementality. When the user adds an alternative to a production, syntactic analysis is performed to check for the alternative's correctness and semantic analyses are performed to update the production and other productions dependent on the production being re-defined. The latter analyses are necessary to ensure that all DFAs affected remain deterministic through updating. Furthermore, if the affected DFAs include the DFA representing the scanner, the updating process is to ensure the overall correctness of the scanner.

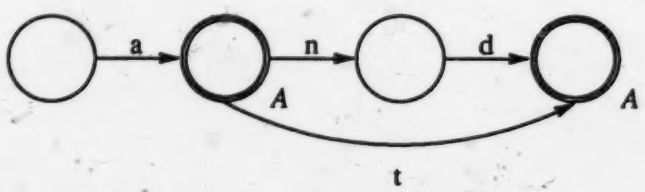
Consider the production, A, which has been defined as <<A>> ::= ab | <B>b and the production, C, which has been defined as <<C>> ::= cb. Note that both A and C are tokens. Suppose the user wants to define B as <B> ::= c. Since A is dependent on B, it is necessary to update the DFA representing A and check for the correctness of the DFA

representing the scanner with the new definition of B. Now, there is a conflict resulting from the fact that when B is expanded in the definition of A, there are two tokens, A and C, which match the string, "cb". This prohibits the token definition of B as A would have been made incorrect. So, the expression, "c", is left in the text-entry view of the special dialog window. Figure 4.5(b) shows the dialog window in which an error message has been inserted before the expression, "c".

4.2.3. Debugging in LexAGen

Graphical interfaces can also have a significant impact on the run-time features of programming environments, especially debugging [Szaf86]. Debugging consists of three major activities: selection, viewing, and modification of the software's internal state. Even though LexAGen is a special-purpose programming environment, debugging is a necessary part of the process of generating a scanner. The graphical user interface is used to enhance the productivity of the user and, therefore, speeds up the process of scanner generation.

In general, a mouse and a bit-mapped display can be used to allow arbitrary programming structures to be selected, executed, and modified. In LexAGen, the user is able to select a DFA for viewing and executing. A selection may be an alternative in the right pane of the browser or a definition of production in the left pane. Two specific menu commands, "display" and "execute", have been provided to display the appropriate DFA in a designated window and to allow the user to test the expressions which it can recognize. Figure 4.6 shows the DFA for a production. If this DFA was executed, then the user would be asked for the input string and a sequence of token values and token strings



where *A* represents the token value to be returned

Figure 4.6. A Typical Deterministic Finite Automaton

are presented to the user in another designated window.

At present, LexAGen displays a textual representation of a DFA instead of a graphical one. Nevertheless, this display has proven to be useful for software testing, especially during the developmental stages of LexAGen itself. Moreover, LexAGen prohibits modification of a DFA directly in the display, since these DFAs are the smallest incremental units. However, it may be possible to relax this restriction by employing a graphics-mode display and extending the smallest incremental unit to a finite state internal to the DFA's structure.

4.2.4. Special Tokens in LexAGen

LexAGen allows the user to designate one (and only one) production as the rule for identifiers. This is done by using a menu in the left pane. After this production has been chosen, a menu selection can now be used to enter keywords which are lexically equivalent to identifiers. Each keyword entered is checked to ensure that it satisfies the rule for identifiers. Keywords which are lexically the same as identifiers but not reserved

(like standard identifiers in Pascal) can be dealt with by the invoking parser (or other user application). They will be treated by the scanner as though they were reserved. On the other hand, keywords which are not lexically the same as identifiers (like Algol's) can be specified as general tokens.

Strings and comments are specified by choosing commands from the left pane menu, and the delimiters are entered in dialog windows. Lookahead for an alternative is specified by choosing a command from the right pane menu, and the lookahead characters are entered in a dialog window. The lookahead characters are then attached to the end of the alternative with the metacharacter, <, prefixing them.

Chapter 5

Implementation of the LexAGen Environment

This chapter describes the implementation of the LexAGen environment. There were two specific design goals that influenced both the functionality of the environment and its implementation. The first goal was to design an integrated software environment. The second goal was to represent a specification by a general, uniform, and incrementally editable data structure which could be translated in a straightforward manner to a compact, efficient scanner.

The first design goal was realized by basing LexAGen on the Smalltalk-80 environment and user interface paradigm. The second design goal was realized by using deterministic finite automata (DFAs) to represent specification of productions, implementing them in Smalltalk-80 and generating a C-code scanner.

5.1. The Influence of Smalltalk

Since LexAGen is implemented in Smalltalk-80, it is necessary to understand something about the Smalltalk-80 environment and language in order to understand the implementation of LexAGen.

5.1.1. The Smalltalk Environment

The most profound influence that Smalltalk-80 has had on the LexAGen environment comes from the structure of the Smalltalk-80 user interface. This structure is

referred to as the model-view-controller (MVC) paradigm [Szaf88]. Although a Smalltalk-80 user is not strictly forced to use this scheme, the support provided makes user interface construction relatively straightforward.

As the terminology suggests, each component of the user interface is divided into three parts. The first part is the model which represents the application in the traditional scheme of UIMS (User Interface Management Systems). The model contains all of the application dependent information and code. The view and controller compose what is traditionally referred to as the user interface. The view component is responsible for displaying information on the screen. The controller is responsible for accepting user input including: the display of pop-up menus, cursor tracking, and mapping user inputs into messages for the application.

The two-pane browser is an example of the MVC paradigm in LexAGen. The model is a data structure which contains production information. There is one view for each pane where the left view displays production names and the right view displays production alternatives. The controller displays the context-sensitive menus by querying the browser for information about the selected structures. It also translates menu selections into messages to the browser, some of which result in the manipulation of individual productions.

5.1.2. The Smalltalk Language

Smalltalk-80 is an object-oriented language [Wegn87]. An object is an abstract entity that consists of two parts: its state and its behavior (the set of messages to which it can respond). The state of an object is represented by instance variables whose values are

other objects. Behavior is specified using the concept of classes. That is, every object is an instance of a class and the behavior of the object is determined by that class. Finally, Smalltalk-80 organises its classes using a tree-structured inheritance mechanism, where classes inherit behavior from their parent classes.

Classed objects which inherit behavior have been used throughout the LexAGen environment. Although objects are used for the user interface and code generation, the most important classes are those representing DFAs. Instances of automata are used to represent components of scanner specifications. Incremental editing of DFA is achieved by defining a set of operations which are implemented as messages. Although the DFA represents components of the specification, it also represents the scanner as well since in the end, it is used to generate C code.

5.2. Data Representation and Implementation

In LexAGen, there are three levels of data representation. The first level implements the user interface and assistance. The second, the heart of the system, represents the scanner specification and ultimately the scanner itself. Finally, the third level is responsible for code generation.

5.2.1. The Browser

Recall that the browser is a model component from the MVC paradigm. The browser is the link between the user and the internal representation of the scanner specification. It maintains a number of internal tables. Two tables are used for the purpose of presenting a consistent user interface, one for the left pane and one for the right.

The other tables are used to access specific properties of the scanner being generated.

In addition, the browser maintains a set of context-sensitive menus, two belonging to the left pane and two to the right. For each pane, there is an initial and a final menu. The initial menu is used when there is no selection in a pane, whereas the final menu is used when a selection has been made.

5.2.1.1. The Left Pane

In the left pane, both the initial and final menus consist of two parts: static and dynamic parts. The static part of the menus contains those commands which always appear, while the dynamic part consists of a composition of context-sensitive components.

The context-sensitive commands are dependent upon the status of the item selected. For instance, whether the name of a production is a token or not. If the name is not a token, then the command "define token" will appear so that it may be specified as a token. If the name is already a token, on the other hand, the command "define non-token" will appear so that its token status may be removed.

The commands are also dependent upon the status of the overall structure. For instance, whether there are tokens already defined or not. If no tokens are defined, then the commands "execute current production" and "execute scanner" will appear; otherwise, only the command "execute scanner" will appear.

Furthermore, these menus are hierarchical menus. That is, when some commands are chosen, submenus appear depending on the context of use. For example, the above two commands "execute production" and "execute scanner" are replaced by this single

command "execute" that when chosen, a submenu containing these two items "current production" and "scanner" are presented to the user.

The first table maintained by the browser is used for displaying the left pane. It is essentially a symbol table implemented as a Smalltalk-80 Dictionary [Gold85]. It maintains all information associated with user-defined production names. Each key is a production name and each value is a named automaton representing all the alternatives for that name. The view for the left pane of the browser simply displays the keys of the table.

5.2.1.2. The Right Pane

The purpose of the right pane is to add, delete, or edit alternatives to the production whose name appears in the left pane. Since there can only be one active context at a time (the production whose name is selected in the left pane), both the initial and final menus in the right pane are static. These static commands correspond to the basic operations that can be performed on each selected production, for example, adding, deleting, editing, displaying, or executing an alternative.

The second table that the browser maintains is for displaying the right pane. It is basically a memory cache implemented as a dictionary, where each key is a literal expression representing the right hand side of a single alternative and each value is a simple automaton representing that alternative. Whenever a name is selected in the left pane, the browser consults the corresponding production for all of its alternatives and records all information about the alternatives (i.e. literal and internal forms) in this table. Whenever a literal expression is selected in the right pane, this memory cache table is used to access the corresponding automaton. For example, if the name <A> had an alternative

b<B>, then the literal "b<B>" would be a key in the memory cache table and its value would be the automaton representing b<B>.

The browser could consult the selected production for the alternatives each time the user makes a selection in the right pane of the browser. However due to the high frequency of activities in the right pane, the increased lookup speed is worth the extra space taken by the memory cache.

5.2.1.3. Other Browser Tables

The browser also maintains other tables to assist the user in developing the final product. First, a table is used to record the names of undeclared productions. That is, non-terminals which have been referenced in productions but do not appear in the left pane.

Another dictionary is used to store information about the values to be returned upon successful recognition of tokens. The keys are token names and the values are the numeric token values to be returned by the generated scanner.

The browser also maintains two tables which serve as specification libraries. The first library contains a collection of productions that are commonly used in programming languages. Some examples are: <digit>, <lowercase letter>, <uppercase letter>, <letter>, and <white space>. This library is read-only, and it provides a quick access to common productions that require extensive enumerations. The second library is a save area for user-defined productions which are general enough that they may be used in more than one scanner. Both of these libraries are implemented as dictionaries with the same structure as the production-name symbol table described previously.

Finally, the browser stores information about the generated scanner in the form of a single grand automaton and maintains a number of special data structures which are used in this scanner. As mentioned in Chapter 4, LexAGen has incorporated certain programming language constructs. Specifically, the following information is maintained: the production which defines identifiers, the set of keywords, comment delimiters, and string delimiters.

5.2.2. The Specification/Scanner

The second level of representation uses deterministic finite automata to represent scanner productions. A collection of specialised classes are used to implement the deterministic finite automata. There are two classes: Automata and AutomataState, along with subclasses of these classes. The Smalltalk-80 class structure of these classes is shown in Figure 5.1.

5.2.2.1. Class Automata

Instances of the class Automata denote deterministic finite automata. Each automaton has two instance variables: lexics and startState. The variable, lexics, refers to the literal form of the underlying automaton. For example, an automaton that recognizes ab<C>, where a and b are terminal characters and <C> is a non-terminal, would be represented by an instance of the class Automata in which the instance variable, lexics, would have the value "ab<C>". Recall from the description on finite automata in Chapter 2, that each automaton has a start state. The instance variable, startState, contains this special state. Instances of the class Automata are used to represent ordinary DFA. For example, a single alternative of a production can be represented by a DFA. On the other

Automata (lexics startState)

NamedAutomata (subAutomata dependents priority selfReferenced)

AutomataState (transitionsTable tokenValue)

LookaheadState ()

Figure 5.1. Class Structure of the Deterministic Finite Automata Model

hand, the entire scanner could be represented by a single instance of the class Automata as well.

Instances of the class NamedAutomata are used to represent complete productions that denote named productions. That is, an instance of NamedAutomata is a structured object which contains many individual automata as alternatives. The class NamedAutomata is a subclass of the class Automata. Instances of NamedAutomata contain four additional instance variables: subAutomata, dependents, priority, and selfReferenced. The most important instance variable is *subAutomata*, which is a collection of all alternative automata making up the named automaton. Note that the inherited instance variable, startState, has as its value the start state of a single automaton, which represents the alternation of all of the sub-automata.

5.2.2.2. Class AutomataState

An instance of the class AutomataState represents a single state in a deterministic finite automaton and has two instance variables: transitionsTable and tokenValue. The instance variable, transitionsTable, is a dictionary of transitions leading out of the state.

The keys of the dictionary are the labels of the directed arcs in the transition diagram and the values are the states to which these transitions lead.

The second instance variable, tokenValue, contains the value of the token which is returned by the generated scanner when the next input character does not correspond to a legal transition or when there is no more input. If the state is not an accepting state, then the tokenValue is the Smalltalk constant, nil.

A subclass of AutomataState called, LookaheadState, is used to implement the two-character lookahead described in Chapter 4. The behavior of this state differs from the class AutomataState in that an input character is examined without being read, and the previously read character is re-inserted into the input stream.

5.2.2.3. The Construction Process for DFAs

In LexAGen, the construction process is performed on the level of states and consists of two operations: concatenation and alternation. Concatenation is a straightforward operation which is performed on the components of each single alternative. Alternation is the process of combining alternatives, and it is the core part of the construction process.

5.2.2.3.1. Concatenation

As an example of concatenation, consider the addition of the alternative, "abc" to a production. A start state and three other states with transitions representing the symbols, a, b, and c, are concatenated to form an automaton as shown in Figure 5.2.

Concatenation is not a batch operation. Concatenation of a single alternative is invoked by the browser iteratively after checking for input correctness. The browser

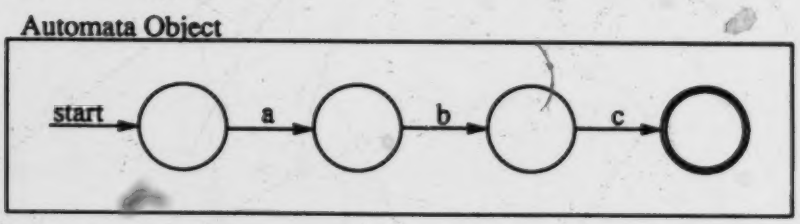


Figure 5.2. Concatenation Operation

parses the input expression into a sequence of syntactically correct labels, either terminals (single-character symbols like a and b) or non-terminals (like <letter> and <digit>). If a syntax error is found, then it is reported immediately. For example, the string "ab<C" contains a syntax error.

If a string is syntactically correct, then a start state is created as the current state. For each label, a new state is created and the current state is connected to the new state using the label as the transition.

5.2.2.3.2. Alternation

The alternation operation is a process of merging two DFAs and optimizing the resulting DFA to collapse common states. Usually, one DFA is a production and the other is a new alternative for that production. First the start states of the two DFAs are merged. Then, the transitions from the start state of the second DFA are added to the start state of the first DFA. If there were transitions whose labels were common to the start states of the two DFAs, then these states must be merged subsequently as well and this

process continues until the two DFAs have been merged. After merging, the result is optimized.

Consider the production A which has been defined as <A> ::= bsd. Suppose the user wants to define an alternative for A which is the expression "bbc". After the alternative has been converted into a DFA, it is merged with the DFA of the production. A successful merging results in a new start state of the production's DFA, while leaving the start state of the alternative's DFA unchanged. Figure 5.3 shows the two DFAs, the merged result and the result of optimizing the merged DFA. Since the start states of both DFAs contained a transition on the common terminal "b", the two states with input "b" were merged. Since the labels of the transitions leaving this newly merged state were "s" and "b", the next two states were not merged. Finally, since the merged DFA contained two identical states (two accepting states with token value A and no transitions), the optimizer merged these two states.

When the transitions of the two DFAs being merged consist only of terminals, the merging operation is quite simple. However if some of the transitions are non-terminals, the process is more complex. It is possible that some of the non-terminals may need to be expanded to prevent the merging process from resulting in a nondeterministic finite automaton. For example, consider the productions <A> ::= ca and <B> ::= cb. If the alternative <A> ::= <B>d is added to production <A> without expanding it, then the result is the nondeterministic finite automaton shown at the top of Figure 5.4. This automaton is nondeterministic since the start state of the merged automaton has transitions on c and <B>, but <B> has c as its first transition. If the non-terminal, <B>, is expanded during the merging process, then the resulting automaton is the deterministic finite automaton

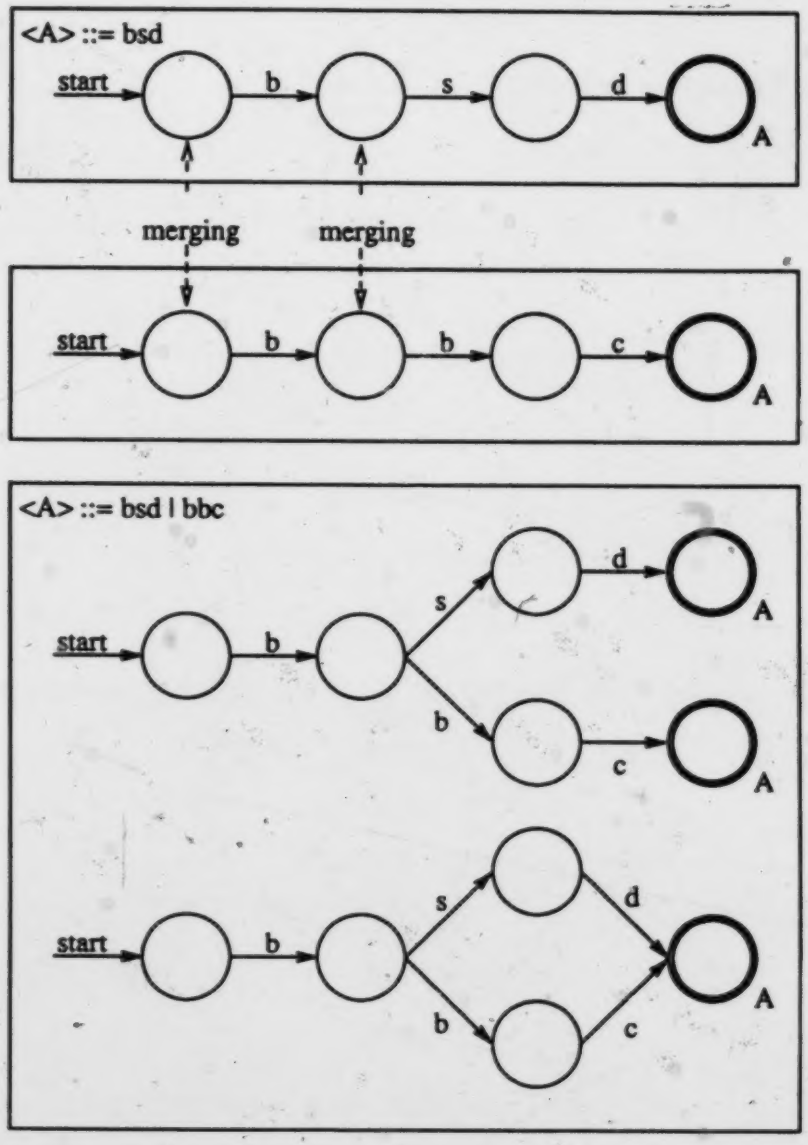
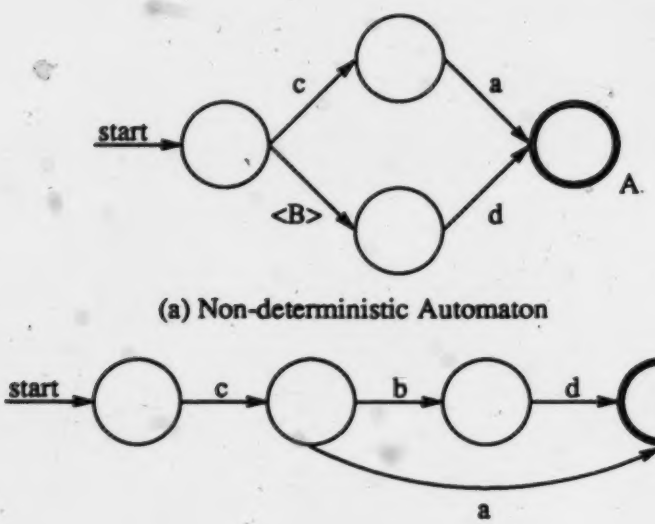


Figure 5.3. Alternation Through Merging



(a) Non-deterministic Automaton

(b) Deterministic Automaton

Figure 5.4. Merging Involving Non-Terminal

shown at the bottom of Figure 5.4.

On the other hand, not all of the non-terminals should be expanded when merging automata, since the expansions reduce the efficiency of editing. For example, consider the production <A> ::= <B>a, where the production <B> is a complex automaton with many alternatives. If <A> was stored in expanded form, then any change to <B> would have to be re-created in <A>. Therefore, in LexAGen, non-terminals are only expanded when necessary.

There is one other consideration when merging two DFAs. If two DFA states represent accepting states and have different token values, then they cannot be merged.

The interface ensures that different alternatives for the same production share a common token value. That is, if the user wants to add an alternative to a production whose name is A, then the token value of A is automatically used for the accepting state generated by the alternative. However, there is a grand automaton which is maintained, and it can contain accepting states with different token values like identifiers and integers. When merging two states in the grand automaton, such a conflict can occur.

For example, consider a production <A> ::= ca with a token value of A and a production <B> ::= cb with a token value of B. If the user tries to add the alternative, <B> ::= ca, then there is no problem in merging the two alternatives for <B>. However, the grand automaton would contain two conflicting states since the string "ca" would be accepted with two token values, A and B. LexAGen detects conflicts during the merging process when the grand automaton is updated as the result of an editing process. If a conflict should occur, then an error is reported to the user and the editing operation is disallowed.

Based on these considerations, the following algorithm is used for merging the states of two DFAs. The set FIRST(X) is defined [Aho86] as the set of terminals that begin strings which are derived from X.

Algorithm 5.1 Merge

Given two states, S_1 and S_2:

[1] If S_1 and S_2 have different token values, then a conflict would exist and so an error is reported. Otherwise, apply step [2].

[2] Add all the transitions of S_2 to S_1, one at a time, according to:

Let T be a transition of S_2 that consists of a label, L (terminal or non-terminal), and a state, S.

(a) If S_1 contains L, then merge S with the state in S_1 connected to L.

(b) If S_1 contains a label, M (terminal or non-terminal), such that

$$FIRST(M) \cap FIRST(L) \neq \emptyset,$$

then:

(i) If M is a dependent of L (i.e. M is defined in terms of L), then expand M in S_1 and try to add T to S_1 again.

(ii) Otherwise, create a new state, R (so that S_2 is not affected), and add T to R. Then expand L in R and merge R with S_1.

(c) Otherwise add T to S_1 since S_1 does not contain L either explicitly or implicitly in a non-terminal.

5.2.3. The LexAGen Strategy for Incremental Analysis

In LexAGen, the incremental analysis is the process by which affected DFAs are updated after changes have been made to the specification. Specifically, incremental analysis involves re-analyses of all dependent DFAs when some DFA is changed. For example, suppose that some non-terminal, A, has been defined in terms of the non-terminals, B, C, and D. Also, suppose that B and C have been defined in terms of the non-terminal, D, respectively. Finally, assume that the user tries to add an alternative to D. Since D changes, all the productions which have been defined directly or indirectly in terms of it must be updated. That is, A, B, and C must be updated. Figure 5.5 shows a

dependency graph for A, B, C, and D. If at any point, an update would result in a conflict in the grand automaton (as described in the previous section), then LexAGen disallows the initial alternative to be added to D and reports an error. Note that as mentioned in Chapter 4, cycles will not exist in the dependency graph used for updating DFAs.

The implementation of this incremental re-analysis is based on dependencies in Smalltalk-80. In Smalltalk-80, an object can be made a dependent of any other object. When an object changes, it sends itself a "changed" message which causes "update" messages to be broadcasted to all of its dependents. Since dependents may have dependents, a graph structure of "update" messages results.

In such a process, it is possible for an object to receive many "update" messages. For example, consider the situation of Figure 5.4. If node D changes, then it will send an "update" message to A, B, and C. Nodes B and C will subsequently send two respective "update" messages to their dependent, A. As a result, A receives two such messages. Such redundant dependencies are common in scanner specifications and could result in slow updating. To solve this problem, LexAGen replaces the standard dependency graph by an ordered dependency graph.

LexAGen uses a two-pass approach to implement the ordering of the dependency graph. Each named DFA maintains an internal priority flag which has an initial value of negative one. A changed DFA sets its internal priority flag to zero and then broadcasts a "pre-update" message to all of its dependents. This "pre-update" message contains a priority parameter which is one greater than the internal priority flag of the sending DFA. When a DFA receives a "pre-update" message, it compares its internal priority flag to the

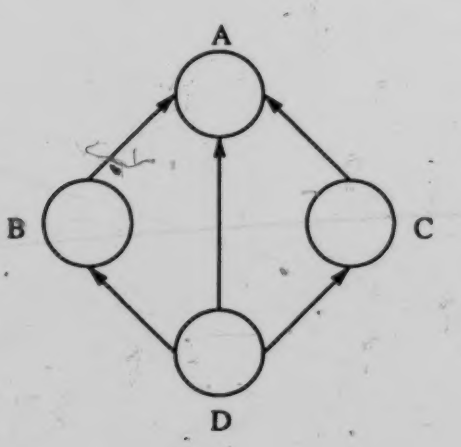


Figure 5.5. Ordered Dependency Graph

priority flag to the priority parameter. If the priority parameter is higher, then it resets its internal priority flag to the value of the priority parameter and broadcasts a "pre-update" message to its dependents with a priority parameter equal to its internal priority flag plus one.

This algorithm orders the dependency graph by assigning to each node, a priority value which is equal to its level in the graph. For example, in the case of the DFAs shown in Figure 5.5, the priorities would be: zero for D, one for C, one for B, and two for A. That is, every DFA has a priority flag whose value is greater than the values of the priority flags of all of the DFAs on which it depends.

Once the first pass has been completed, the DFAs are updated in a breadth-first order. That is, the original DFA broadcasts an "update" message which asks its dependents to update themselves according to their priority counts from lowest to highest. The

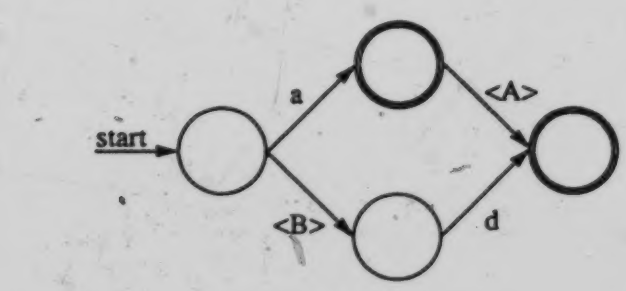
root DFA broadcasts an "update" message with priority one. All dependents with internal priority flag values of one, update themselves and broadcast an "update" message with priority two. This continues until all dependents have been updated. Notice that each dependent is only updated once and that the update occurs after all of the productions on which it depends.

5.2.4. The Coder

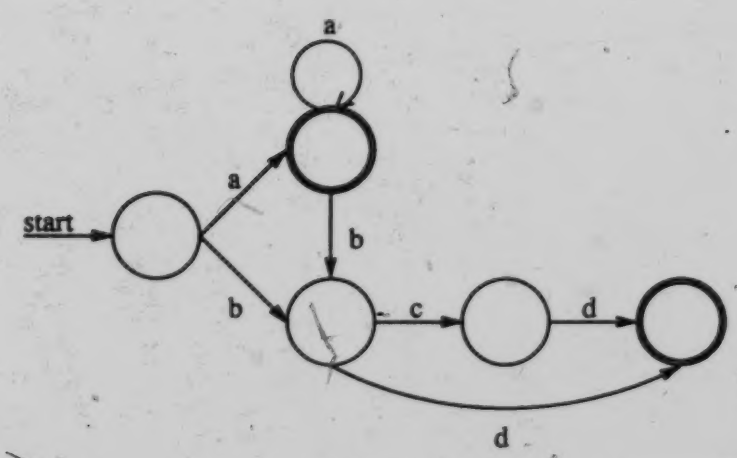
The third level of representation is the coder, which is responsible for code generation. When the user requests the code for a specification, LexAGen asks the grand automaton to expand itself and then translates the expanded automaton into intermediate code. The intermediate code is then translated into the target language. Figure 5.6 shows the transition diagrams for an automaton A and the expanded form of the automaton A. Note that A is defined as <A> ::= a | a<A> | <B>d and B is defined as <B> ::= b | bc.

The coder is implemented by the class AutomataCode and its nine subclasses. An instance of the class AutomataCode has one instance variable, operations, which represents a stack of operations to be performed. Each subclass represents a single kind of operation to be translated into the target language. Instances of these subclasses represent the intermediate code. The Smalltalk-80 class structure of the class AutomataCode is shown in Figure 5.7.

Currently the target language is C, but coders for other languages can be constructed easily. This can be done by changing the behavior of the subclasses which represent the intermediate code, so that they are translated into a different target language. Figure 5.8 shows the executable C-program produced by the coder for the automaton shown in



<A> ::= a<A> | <B>d



Expanded Form of A

where <B> has been defined as <B> ::= b | bc

Figure 5.6. Full Expanded Form of a DFA

Figure 5.6. This program is composed of nested case statements which resemble the exact structure of the automaton being generated.

In fact, the coder generates a complete module for this program with its own program interface (name is given by the user). In addition, the coder produces a file of token values as an include file to the module so that this include file may be used by other

AutomataCode (operations)

AutomataAdvance ()

AutomataBreak ()

AutomataBackup ()

AutomataCase (labels)

AutomataDefault ()

AutomataGoto ()

AutomataLabel ()

AutomataSwitch ()

AutomataToken ()

Figure 5.7. Class Structure of the Coder

application programs (for instance, a parser). A library is also provided to support the generated scanner. For example, the library incorporates a specialised buffering technique for reading input characters. Furthermore, the library maintains a token buffer to store the sequence of characters in the source program that is matched by the pattern for a token. So, a client program is provided with two kinds of information upon each successful recognition: a token and its corresponding string (semantic value).

This approach of nested case statements can sometimes result in (time) inefficient code [Wait86a]. Inefficient code is generated when a compiler translates these case statements into machine code as if-then-else statements instead of as jump tables. Most com-

```
switch (*p) {   /* examine current character */
    case 'a' :
        LABEL0:
        switch (*++p) { /* examine next character */
            case 'a' :
                goto LABEL0;
            case 'b' :
                switch (*++p) { /* examine next character */
                    case 'c' :
                        switch (*++p) { /* examine next character */\
                            case 'd' :
                                ++p; /* advance to next character */
                                return(A); /* accept token */
                        } --p; break; /* backup by one character */
                    case 'd' :
                        ++p; /* advance to next character */
                        return(A); /* accept token */
                } --p; break; /* backup by one character */
        }
        return(A); /* accept token */
    case 'b' :
        switch (*++p) { /* examine next character */
            case 'c' :
                switch (*++p) { /* examine next character */
                    case 'd' :
                        ++p; /* advance to next character */
                        return(A); /* accept token */
                } --p; break; /* backup by one character */
            case 'd' :
                ++p; /* advance to next character */
                return(A); /* accept token */
        } break;
    default : ++p; /* advance to next character */
}
return(NIL); /* accept no token */
```

Figure 5.8. Sample C Source Codes Generated

pilers will not use jump tables if the character codes of the labels in a case statement are widely separated. For example, a case statement involving all of the letters from 'a' to 'z' will be translated into a jump table, while a case statement involving the letters '#',

An alternative approach (which is still in the category of direct execution) involves the use of character tables. Each table encodes integer values for each character of the character set in use (for instance, the ASCII character set). The table values are assigned so that characters in the same case clause have the same values. These values are used in a case statement which is sure to be translated into a jump table, since the values are chosen continuously. For example, the table may map all of the characters, 'a' to 'z' and 'A' to 'Z', to a single value, say, 1 representing the first character of an identifier. It may also map all of the characters, '0' to '9', to a single value, say, 2 representing the first character of a number.

The disadvantage of using character tables is that an extra level of indirection is required (the table lookup) and this can slow down the scanner considerably. A better approach is to use case statements and force the compiler to implement them with jump tables. If a compiler does not have a switch which can be set to do so, a scanner generator can explicitly generate cases for all ASCII character codes to force the compiler to use a jump table implementation. Currently, LexAGen does not generate a full set of cases, but it can easily be modified to do this.

Chapter 6

Keyword Identification

Keyword identification is the process of searching a list of pre-defined keywords to determine if a general identifier is really a language keyword. As was stated in Chapter 4, LexAGen generates a scanner which distinguishes keywords from general identifiers. This chapter discusses keyword identification in general and presents the keyword identification algorithm developed for LexAGen.

Searching a list of data items has always been a ubiquitous activity and the literature is full of search algorithms. However, most efficient search algorithms take advantage of specific attributes of the set of data items being searched. In the case of keyword identification, the efficiency of such searches is dependent on the specific keywords for the language.

Hand-coded lexical analyzers can be easily tailored to take advantage of the attributes of a particular set of keywords. However, lexical-analyzer generators must be capable of generating scanners for languages with different sets of keywords. Various solutions to this problem have been proposed. For instance, postscan hashing [Hors87] and preloaded symbol table [Wait86a] have been used. The problems with postscan hashing are described in [Sebe85]. Essentially, the problem is to find a minimal perfect hash function in terms of collisions and table size. The main problem with preloaded symbol tables is that no logical distinction is made between a user-defined identifier and a language keyword.

In this chapter, a new approach to keyword identification is presented. Given the set of keywords for a language, a data structure called a pruned O-trie forest is constructed. The forest is then used to generate code which identifies the keywords for that specific language. This chapter presents a dynamic programming construction of an optimal pruned O-trie forest. LexAGen uses this algorithm to produce keyword identification code to serve as the last phase of its generated scanners.

6.1. Overview of the Trie-Based Method

A trie-based method, generally speaking, is an indexing scheme that views an alphabetic key as composed of a sequence of characters. In essence, a trie-based method resembles closely to a key-comparison based method or a B-tree search scheme [Baye72]. However, trie-based methods do not rely on the notion of comparing whole keys in constant time. As a consequence, comparison of keys is no longer the elementary operation for standard measures of complexity.

For example, the length of a key can be used as an attribute so that a set of data items can be split into subsets of equal-length keys. Individual characters in fixed positions can also be used as attributes. For instance, the keys "DIV" and "DO", can be distinguished in the second position using the characters 'I' and 'O'. Of course, each fixed position chosen is regarded as a different attribute.

The k-ary tree structure created by successively dividing keys into smaller sets using different attributes is called a *trie* (pronounced as *try*) [Bria59] [Fred60].

6.1.1. Special Tries

Given a set of data items, where k is the maximum of the lengths of the data items, a *full trie* is a tree of depth no more than k such that:

(1) All paths from the root to the leaves are distinct.

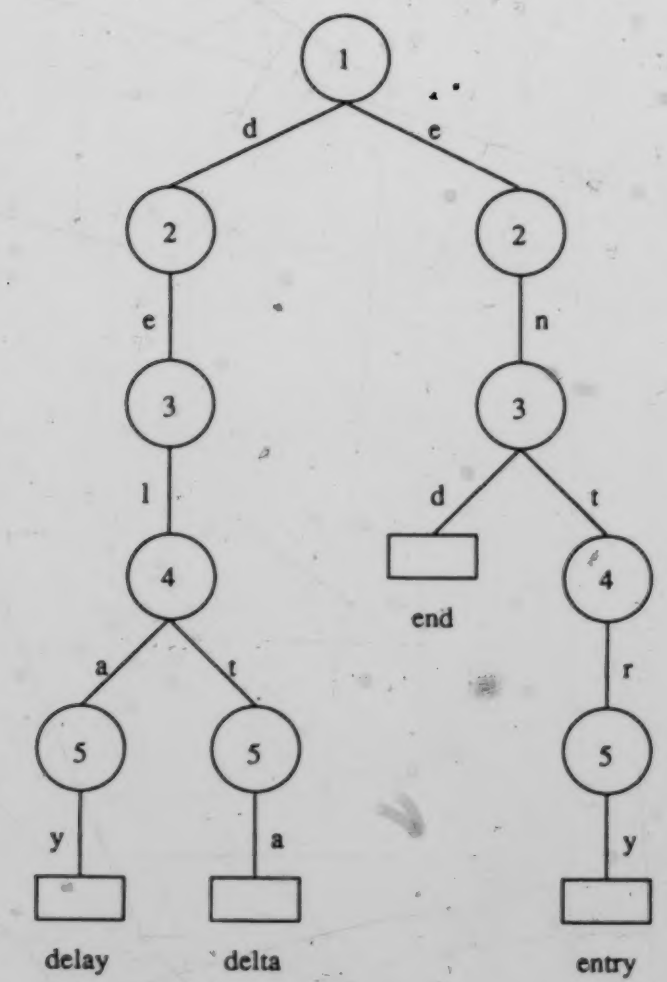
(2) There is a unique path from the root to each leaf node corresponding to an item in the set.

An example of a full trie is shown in figure 6.1 for the reserved words, delay, delta, end, and entry in Ada.

A *pruned trie* is a full trie with no redundant non-leaf nodes along any leaf chain. Redundant non-leaf nodes are those consecutive single-successor nodes that lead to a leaf. Figure 6.2 contains a pruned trie.

A *pruned O-trie* is a generalisation of a pruned trie where different paths from the root may use different attributes. Figure 6.3 shows a pruned O-trie in which the path from the root to the leaf labeled "delay" uses character positions 1 and 4, while the path from the root to the leaf labeled "end" uses character positions 1 and 3.

In the LexAGen environment, the length-of-key attribute is used to divide the keys into sets of equal lengths and then character positions are used for the rest of the attributes. To increase efficiency, different paths use different positions. To support this algorithm, a *pruned O-trie forest* is defined as a collection of pruned O-tries with the property that each pruned O-trie represents a subset of equal-length data items.



where numbers in circle indicate character positions

Figure 6.1. A Full Trie

6.1.2. Example of a Pruned O-Trie Forest

Consider the following reserved words from Ada:

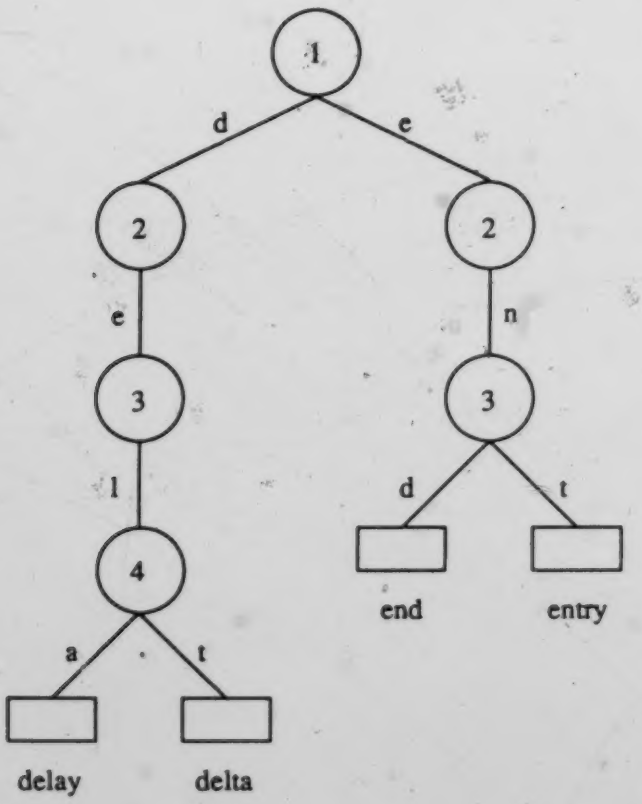


Figure 6.2. A Pruned Trie

delay, delta, entry, if, in, of, raise, range.

These keys can be divided into two subsets, where one subset consists of keys of length two and the other contains keys of length five. As shown in Figure 6.4, the tree of length-two keys uses the attributes: character position one and then character position two to produce a pruned trie. The tree composed of length-five keys uses the attribute: character position one and then either character position three or character position four,

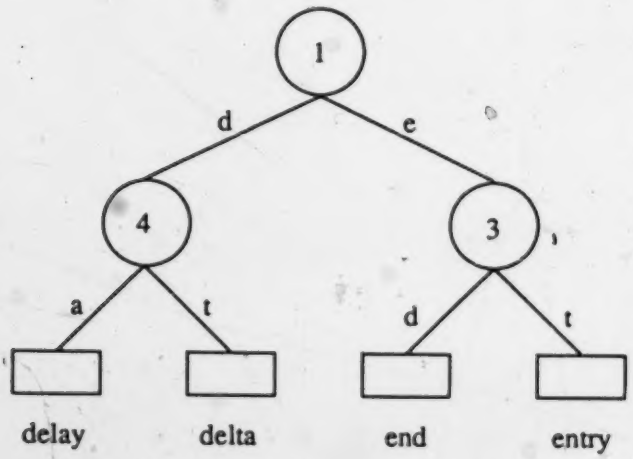


Figure 6.3. A Pruned O-Trie

depending on the path chosen.

Thus instead of testing all characters or attributes in a string, the search actually follows along a pruned chain that leads to a leaf node [Come76]. An additional string comparison is then done to verify that the string matches the contents of the leaf node. Notice that in the example used in Figure 6.4, a tree of smaller depth (depth one) could have been constructed for the keywords of length five if character position four had been used as the first attribute instead of character position one. The algorithm presented in this chapter finds a tree of *minimal cost* (or depth) by using some heuristics to improve speed.

6.1.3. Searching Time for a Pruned O-Trie Forest

Looking for an identifier in a pruned O-trie forest consists of identifying the pruned O-trie corresponding to the length of the identifier and searching the trie on a path from

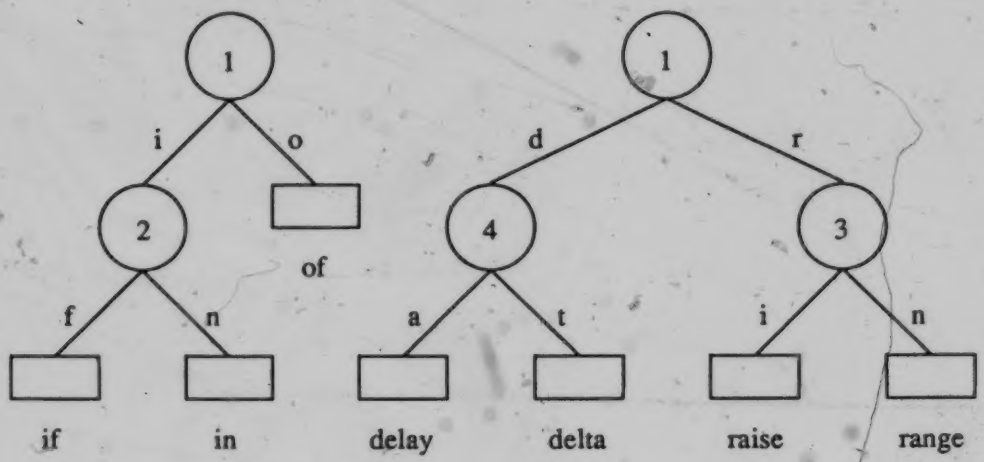


Figure 6.4. A Pruned O-Trie Forest

the root to a leaf. If it is assumed that the correct pruned O-trie can be located in the forest in constant time (say, using a jump table implementation of a case), then it is straightforward to show that the worst-case searching time for a pruned O-trie forest is of the order of the maximum length of the keywords.

Lemma 1 The depth of a pruned O-trie for a set of n-length keywords is at most n, where $n \geq 1$.

Proof: (Induction)

(1) Basis: $n = 1$.

There is only ONE internal node that leads to the leaf nodes containing the keywords, so the depth is 1.

(2) Induction step: Assume true for $n \leq N$ (some constant).

For length of $n + 1$, the worst case is that one of the internal node next to the root node is a n-node subtrie. Since the depth of the n-node subtrie is at most n, the depth of the $(n + 1)$-node trie is at most $(n + 1)$.

Lemma 2 The worst-case searching time of a pruned O-trie for a set of n-length keywords is $O(n)$.

Proof: The searching time of a pruned O-trie is bound by the depth of the pruned O-trie. But this depth is at most n by Lemma 1, and n is $O(n)$.

Lemma 3 The worst-case searching time of a pruned O-trie forest is

$$O(max\{m_1, ..., m_k\})$$

where $m_1, ..., m_k$ are lengths of data items of k equal-length subsets.

Proof: By Lemma 2, it takes $O(m_i)$ time to find a keyword in a pruned O-trie, where m_i is the length of keywords in the pruned O-trie.

6.2. Trie Index Construction

A branch-and-bound algorithm has been devised to construct a pruned O-trie forest for an arbitrary collection of keywords. This algorithm is designed to be used as the last phase of lexical analysis.

Since the trie-index construction problem is NP-complete [Come76], this algorithm is not necessarily fast. That is, this algorithm may take exponential time. However, this

algorithm always returns a minimal-cost (minimal-depth) pruned O-trie forest. If it is assumed that the probabilities of all keywords are equal, then the probabilities of all leaf nodes in the forest are equal such that this algorithm yields an optimal forest (i.e. a forest which generates the most efficient search).

Note that in this algorithm the *cost* of a pruned O-trie is defined as the sum of the depths of each of its leaves.

6.2.1. Algorithm for Pruned O-Trie Construction

[1] Split the set of keywords into subsets of equal-length data items.

Example: Consider the keywords for a hypothetical language, H.

[2] For each of these subsets, apply step [3] with depth equal to 1.

Example: Consider the keywords from H of length 3: {and, end, mod, eot, any}

[3] Given a set of n, m-length data items at depth d, create a decision table with m entries corresponding to the characters in positions 1 to m. The decision-table entry at location i is a dictionary whose keys are all of the i^{th} characters from all of the n data items. The value of each key at location i is the subset of data items which contain the key character at location i. If the number of entries in any one dictionary is equal to n, then return to the calling step with a three-component record containing: the value of i, a cost which is the product of n and d, and the i^{th} dictionary. Otherwise, go on to step [4] and [5], and return the results to the calling step.

Example: Decision table (n = 5, m = 3, d = 1):

<i=1, [(a, {and, any}), (e, {end, eot}), (m, {mod})]>

<i=2, [(n, {and, any, end}), (o, {eot, mod})]>

<i=3, [(d, {and, end, mod}), (t, {eot}), (y, {any})]>

Since none of the dictionaries contains a five-element value set, go on to step [4].

[4] From the decision table, create two control stacks, A and E, which store records of the form defined in [3], where the cost is either an actual minimum cost or an expected minimum cost. Both stacks are sorted by cost in ascending order (i.e. the top-of-stack record corresponds to the pruned O-trie with the least cost value among all the tries in the stack). The cost of the i^{th} dictionary is defined to be the sum of the individual costs associated with the data items in its value sets. If the size of a value set is equal to one, then its cost is equal to the depth, d; otherwise, its cost is equal to the product of its size and (d + 1). If the size of any value sets is greater than two, then the cost is an expected minimum cost; otherwise, the cost is an actual minimum cost. Records which contain actual costs are pushed onto the A stack, and records which contain expected costs are pushed onto the E stack. As the algorithm proceeds and expected costs are refined into actual costs, the E stack will shrink and the A stack will grow. Go to step [5].

Example: The A stack contains one record with contents:

<i=1, C=2*2+2*2+1=9, [(a, {and, any}), (e, {end, eot}), (m, {mod})]>

The E stack contains two records with contents:

<i=3, C=3*2+1+1=8, [(d, {and, end, mod}), (t, {eot}), (y, {any})]>

<i=2, C=3*2+2*2=10, [(n, {and, any, end}), (o, {eot, mod})]>

[5] a) If (Stack E is empty) or (Stack A is non-empty and the top-of-stack record in Stack A has a cost which is less than the cost in the top-of-stack record in Stack E), then return the top-of-stack record from A to step [3].

b) Otherwise, pop Stack E, and replace its cost and dictionary components using [6]. Push the resulting record onto the A stack since the cost will now be actual. Repeat step [5].

Example: a) does not apply since the top-of-stack record for Stack A has cost 9 and the top-of-stack record for Stack E has cost 8; so use b).

Step [6] returns the replacement record:

<i=3, C=8, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>.

So, Stack A contains the records:

<i=3, C=8, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>.

<i=1, C=9, [(a, {and, any}), (e; {end, eot}), (m, {mod})]>

The E stack contains one record with contents:

<i=2, C=10, [(n, {and, any, end}), (o, {eot, mod})]>

Now, a) applies since the top-of-stack record for Stack A has cost 8 and the top-of-stack record for Stack E has cost 10; so return the top-of-stack record from A to step [3].

[6] Reset the cost component of the current record to zero. Apply step [3] to each value set of the current record whose cardinality is greater than one. Use a depth of d + 1. Remove the cost component of the record returned by step [3] and add the cost component to the cost component of the current record. Replace each value set in the current record by the record returned by step [3] with the cost component removed. Do not replace any of the value sets of size one. However, add d to the cost component of the current record for each value set of size one. Return the modified record to step [5].

Example: Reset the cost component of the current record to zero:

<i=3, C=0, [(d, {and, end, mod}), (t, {eot}), (y, {any})]>

The current record contains one value set of size greater than one: {and, end, mod}. So, apply step [3] to this set with a depth of 1 + 1 = 2. The record for this set returned by step [3] is:

<i=1, C=3*2=6, [(a, {and}), (e, {end}), (m, {mod})]>

Remove the cost component from the above record and add to the cost component of the current record; also, replace the value set by the record returned by step [3] with the cost component removed:

<i=3, C=6, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>.

The current record contains two value sets of size one: {eot} and {any}. So, add 2*1 = 2 to the cost component of the current record and return the modified record to step [5]:

<i=3, C=8, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y,

{any})]>.

6.2.2. Implementation

The LexAGen environment produces C code which implements the optimal pruned O-trie forest produced by the algorithm. Each non-leaf node is represented by a case statement of character labels, where the labels are those characters which identify the branches in the trie. Leaf nodes are represented by case statements which correspond to those character labels not previously appeared in the non-leaf nodes along any leaf chain.

The code for the optimal pruned O-trie for the Modula-2 reserved words of length two:

BY DO IF IN OF OR TO

is shown in Figure 6.5.

Putting aside the complexity of the construction algorithm, an important observation about the resulting minimal-depth pruned O-trie forest is that in theory, its access time is independent of the number of keywords and depends linearly on the maximum length of the keywords. In practice, however, the average depth of a minimal-depth pruned O-trie forest is considerably smaller than the maximum length of the keywords, especially given a large length. In fact, the depth is usually one or two. Table 6.1 shows the sizes of the sets of equal-length keywords and the depths of the corresponding pruned O-tries for several programming languages.

```
switch (strlen(aString)) { /* length of the string */
    case 2 :
        switch (aString[0]) {
            case 'B' :
                switch (aString[1]) {
                    case 'Y' :
                        return(BY); }
                return(IDENTIFIER);
            case 'D' :
                switch (aString[1]) {
                    case 'O' :
                        return(DO); }
                return(IDENTIFIER);
            case 'I' :
                switch (aString[1]) {
                    case 'F' :
                        return(IF);
                    case 'N' :
                        return(IN); }
                return(IDENTIFIER);
            case 'O' :
                switch (aString[1]) {
                    case 'F' :
                        return(OF);
                    case 'R' :
                        return(OR); }
                return(IDENTIFIER);
            case 'T' :
                switch (aString[1]) {
                    case 'O' :
                        return(TO); }
                return(IDENTIFIER);
        }
    return(IDENTIFIER);
}
return(IDENTIFIER);
```

Figure 6.5. Implementation of a Pruned O-Trie

	Pascal		Modula-2		Ada	
length	size	depth	size	depth	size	depth
2	6	2	7	2	7	2
3	9	2	8	1	12	2
4	7	2	8	1	12	2
5	6	1	6	1	10	2
6	4	1	6	2	8	2
7	2	1	1	0	8	2
8	1	0	-	-	3	1
9	1	0	2	1	3	1
10	-	-	1	0	-	-
11	-	-	-	-	-	-
12	-	-	-	-	-	-
13	-	-	-	-	-	-
14	-	-	1	0	-	-

Table 6.1. Characteristics for Several Programming Languages

Chapter 7

A Comparison of LexAGen with Other Scanner Generators

This chapter evaluates the scanner generators mentioned in this thesis: Lex, GLA, Mkscan, and LexAGen. Table 1 summarizes the advantages and disadvantages of the four scanner generators based on the three criteria: generality, user interface design, and speed.

7.1. Evaluation of Lex

Lex is capable of generating general scanners. It supports full regular grammars and provides a general scheme of extensions. The generality of Lex can be seen in the freedom Lex allows its users to exercise through regular expressions. The multiple-character lookahead scheme adds to Lex some degree of context sensitivity which is useful for modern programming languages.

However, this capability carries a price. Firstly, users who want to generate a simple scanner, and do not need the generality of Lex, are presented with large scanners. For example, a simple scanner for identifiers and integers is usually translated into several hundred lines of C source code which executes quite slowly when compared to an equivalent hand-coded scanner which requires much less than a hundred lines of C source code.

Secondly, most users who need to specify only simple patterns have to learn the full, general expression notation used by Lex. An apparently simple pattern, at least when

	Generality	User Interface Ease of Use	Speed
Lex	++	-	-
GLA	-	-	++
Mkscan	-	+	+
LexAGen	+	++	+

Table 7.1. Summary of the Four Scanner Generators

stated verbally, may be translated into a relatively complicated regular expression. The regular expression for a C comment is a good illustration of this. The Lex regular expression for a Comment is as follows:

"/*" [^*]* ("*" [^*]*)* "*/"

where it is permissible for a comment to contain the character '*' or '/' as long as they do not appear together and form the comment delimiter.

Another shortcoming of Lex is that it is batch-oriented. User interaction is virtually nonexistent. That is, the user must supply the entire scanner specification before Lex is invoked to look for errors. This, coupled with poor diagnostic capabilities, makes Lex relatively hard to use by novices.

However, it should be recognized that Lex is a first generation compiler generation tool. Despite the inherently slow execution of its generated scanners, Lex was an experiment that was successful and it paved the way for more modern scanner generators.

7.2. Evaluation of GLA

GLA was designed to generate fast scanners for programming languages, and this was achieved by sacrificing generality. It only supports a subset of regular languages plus some limited extensions. To achieve this, GLA imposes constraints on the allowable symbol sets for its specifications. Users make use of a collection of built-in concepts (integers, floating point numbers, comments, strings, identifiers, and so on). Because of these restrictions, GLA generates scanners which are fast.

Unfortunately, GLA is too restrictive to specify many of the tokens in standard programming languages. For example, consider floating numbers. GLA allows the user some freedom in specifying a floating point number (the initial character set, the continuation character set, and so on). Surprisingly however, GLA does not provide enough freedom to correctly specify floating point numbers in Pascal or Modula-2. GLA requires that the decimal point for floating numbers be preceded by an initial or continuation character and that it be followed by a continuation character.

For example, GLA does not allow the user to specify that "12e5" is a legal Pascal floating point number. In addition, GLA requires that "12.12e" is a legal floating point number in all languages, if the character 'e' has been designated as the exponent character. But this is not a legal floating point number in Pascal or Modula-2. Finally, GLA is incapable of specifying that "12." is a legal floating point number, even though this is the case in Modula-2.

As a second example, GLA does not allow C-style hexadecimal constants like "0x123" to be specified. As a third example, GLA does not support the nested comments

in Modula-2. Even though GLA could be modified to support all of the tokens in all existing programming languages, there is nothing to prevent future languages from using reasonable constructs which GLA could not support. The problem is that a scanner generator needs a mechanism for specifying general tokens.

In addition to the token restrictions, GLA is not flexible or easy to learn and use. It uses a batch-oriented, high-level specification language. Along with the specification language, users must also learn the functions of a collection of supporting modules which are incorporated with the generated scanner. The symbol table module is an example of such a supporting module. This is satisfactory for users who are generating scanners which are to be used with parsers generated by traditional parser generators. However, it is too complicated for users who are generating stand-alone scanners or scanners to be used with non-traditional parsers (say, incremental parsers) where the supporting modules may be redundant or simply not required. Like Lex, user interaction is nonexistent.

7.3. Evaluation of Mkscan

Like GLA, Mkscan is intended for programming language scanners. This means that Mkscan also supports a subset of regular languages plus some limited extensions. It imposes certain restrictions on the scanners it can generate, and it is especially biased towards the family of Pascal-like languages. As an example of its restrictiveness to programming language scanners, Mkscan cannot generate a scanner if the user does not specify identifiers (unfortunately, no warning or error messages are reported in this case).

Also like GLA, Mkscan has some restrictions which prevent standard languages from being specified. For example, Mkscan requires that all keywords be valid identifiers. This prevents the generation of scanners which differentiate keywords by case or a special prefix symbol (most Algol compilers). Nested comments are also disallowed, so a correct Modula-2 scanner cannot be generated.

Mkscan advocates that the notion of a regular expression is just one particular way of conceptualising tokens and therefore is more than a part of the implementation than of the specification. However, Mkscan employs certain metacharacters and a pattern notation that is equivalent to regular expressions. For example, the digit, 9, can be used to represent any decimal digit in the range 0 to 9 and the pattern, 99, describes a sequence of one or more decimal digits. One current deficiency is that the user must type many patterns to describe all variations of a token specification. This can be rather taxing in some cases. For example, the specification for Pascal floating point numbers must include:

99.99 99e 99.99e99 99.99e+99

where the character '+' means plus or minus.

Mkscan is very easy to use since it uses full-screen interactive menus to guide the user through a series of choices and in most cases gives immediate feedback as early as possible. Yet, Mkscan does not support full-scale user interaction. For instance, the amount and form of feedback when errors are made is minimal in Mkscan. Furthermore, Mkscan does not support immediate execution even though it could utilise its full screen user interface to do so.

7.4. Evaluation of LexAGen

LexAGen is the result of an attempt to provide a scanner generator which is easy to use and which generates fast scanners for general specifications. The generality of LexAGen comes from its ability to support the full set of regular languages plus some general extensions which are sufficient for most modern programming constructs. LexAGen is also unique among other scanner generators in applying many benefits of graphical user interfaces to scanner generation. For instance, LexAGen is the first (and only) scanner generator that incorporates incremental development. Furthermore, LexAGen provides full-scale user interaction ranging from immediate error reporting to specification execution.

Nevertheless, LexAGen is not perfect. Although its graphical interface makes LexAGen easy to use, its generality requires that the users be familiar with BNF grammar notation. This takes some practice. For example, consider the following BNF definition:

<A> ::= a | a<A> | b | b<A>

where A is recursively defined in terms of itself. Novices may incorrectly interpret this specification as "one of more a's or one of more b's". However, the correct interpretation should be "one or more a's or b's intermixed".

Another shortcoming of LexAGen is its speed of incremental analysis. Although LexAGen is capable of performing incremental construction of DFAs and detecting any state conflicts at the time of definition, it takes time. Since the Smalltalk-80 system in which it is implemented is not a multi-processing system, the user must wait duing these incremental processes.

7.5. Speed Analysis

As far as relative sizes and speeds of generated scanners are concerned, LexAGen has achieved its goal to produce general and efficient scanners. In an experiment, a scanner for Pascal was created using LexAGen, GLA, Mkscan, and Lex. The four scanners were compiled on a SUN 2/50 under the 4.2BSD UNIX system. When compiled, the object code sizes for the four scanners are 9.07K, 24.98K, 4.68K, and 10.74K bytes, respectively.

The large size of the GLA-generated scanner is attributed to the existence of supporting modules. It should also be noted that both LexAGen and Lex include the keyword identification algorithm mentioned in Chapter 6 as part of the generated scanners, and this accounts for the extra size when compared to the Mkscan scanner which uses hashing technique to distinguish keywords from identifiers. The Lex scanner was constructed in such a way that all keywords are recognized as identifiers and handled separately by the module for keyword identification.

Scanner timings were obtained by tokenising one large Standard Pascal program (319.63K bytes). Table 7.2 summarizes the characteristics of the input data. All tests were made under the 4.2BSD UNIX system on the same machine where the scanners were compiled. Although it was run in multi-user mode during the experiment, timings were taken only when no other users were logged on.

Each test was run 10 times, and the mean of those samples was used. The time was the sum of the total amount of time spent executing in user mode and system mode (executing system calls for the scanners). The execution speed ranking of the generated

	Occurrences	
Single spaces	23,700	
Newline characters	12,000	
Identifiers	11,700	(6.38)
Keywords	7,900	(3.78)
;	4,000	
Integers	2,300	(4.7)
(	2,000	
)	2,000	
Space pairs	1,900	
:=	1,800	
Comments	1,600	(81.56)
:	1,500	
,	1,100	
Space triples	1,000	
Reals	900	(15.67)
=	800	
Strings	700	(10.0)
<	600	
-	400	
*	300	
/	300	
..	300	
[	300	
]	300	
+	200	
^	200	
.	100	
<>	100	
<=	100	
>=	100	
>	0	

where numbers in paratheses indicate average lengths in characters.

Table 7.2. Characteristics of the Input Data

scanners is: GLA, LexAGen, Mkscan, and Lex. The ratios of LexAGen, Mkscan, and Lex to GLA are 1.04, 1.10, and 2.69, respectively.

Chapter 8

Conclusion

Scanner generators are still seldom used in practice, because hand-coded scanners are slightly more efficient than generated ones. However, the advantages of generated scanners should not be ignored. Scanner generators can construct correct scanners quickly. The input to a scanner generator is a precise specification and serves as good documentation of the generated scanner. For prototyping, or on fast computers, where small savings in execution time are not of paramount importance, scanner generators may be very useful.

Current scanner generators are either too slow or not general enough for generating programming language scanners. In either case, they are often difficult to use and provide poor error reports. LexAGen is an easy to use scanner generator with immediate error reporting which can generate efficient scanners for programming languages. In fact:

(1) LexAGen generates lexical analyzers which are almost as fast as those generated by GLA, which are programming language specific.

(2) LexAGen supports full regular grammars and enough extensions for modern programming languages.

(3) LexAGen has an outstanding graphical user interface and is the only scanner generator which generates scanners incrementally.

There are two areas of LexAGen that could be enhanced: faster incremental analysis

and multi-character lookahead capability. In addition, as mentioned in Chapter 4, LexAGen currently displays its DFA textually. A graphical display of DFAs could provide an alternative specification technique. The user could manipulate DFAs directly instead of using BNF notation. This would make LexAGen even easier to use.

Furthermore, LexAGen could utilise different fonts of text to display the names of productions in the left hand pane of the browser, for example, italic for non-tokens and bold for tokens. Also, LexAGen may allow the user to enter non-terminals in the right pane through special control sequences or menu commands, and display them in italics.

Finally, since LexAGen is the first component of an envisioned integrated compiler generation environment, other tools like symbol table generator, an error reporting generator, and a parser generator must be built.

References

[Aho86] Aho, A.V., R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

[Baye72] Bayer, R. and E. McCreight, "Organisation and Maintenance of Large Ordered Indexes," *Acta Informatica*, **1**, 173-189 (1972).

[Bria59] de la Briandais, R., "File Searching Using Variable Length Keys," *Proc. WJCC*, 295-298 (1959).

[Cich80] Cichelli, R.J., "Minimal Perfect Hash Functions Made Simple," *Communications of the ACM*, **23**(1), 17-19 (1980).

[Come76] Comer, D. and R. Sethi, "Complexity of Trie Index Construction (extended abstract)," *Proc. 17th Ann. Symp. on Foundations of Computer Science*, IEEE Computer Society, Long Beach, CA, 197-207 (1976).

[Coop83] Cooper D., *Standard Pascal: User Reference Manual*, W. W. Norton, NY (1983).

[Deli84] Delisle N.M., D.E. Menicosy, and M.D. Schwartz, "Viewing a Programming Environment as a Single Tool," *ACM SIGPLAN Notices*, **19**(5), 49-56 (1984).

[Denn78] Denning P.J., J.B. Dennis, and J.E. Qualitz, *Machines, Languages, and Computation*, Prentice-Hall, Englewood Cliffs, NJ (1978).

[Fred60] Fredkin, E., "Trie Memory," *Communications of the ACM*, 3(9), 490-499 (1960).

[Gieg79] Giegerich R, "Introduction to the Compiler Generating System MUG2," *TUM-INFO 7913*, TU München, 1979.

[Gold84] Golderg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA (1984).

[Gold85] Golderg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA (1985).

[Goug88] Gough, K.J., *Syntax Analysis and Software Tools*, Addison-Wesley, Sydney (1988).

[Heur86] Heuring, V.P., "Automatic Generation of Fast Lexical Analyzers," *Software — Practice & Experience*, **16**(9), 801-808 (1986).

[Hors87] Horspool, R.N. and M.R. Levy, "Mkscan — An Interactive Scanner Generator," *Software — Practice & Experience*, **17**(6), 369-378 (1987).

[John75] Johnson, S.C., "Yacc — Yet Another Compiler-Compiler," *Computing Science Technical Report 32*, Bell Telephone Laboratories, Murray Hill, NJ (1975).

[Lesk75] Lesk, M.E. and E. Schmidt, "Lex — A Lexical Analyzer Generator," *Computing Science Technical Report 39*, Bell Telephone Laboratories, Murray Hill, NJ (1975).

[Moss86] Mössenböck, H., "Alex — A Simple and Efficient Scanner Generator," *ACM SIGPLAN Notices*, **21**(5), 69-78 (1986).

[Nurm82] Nurmi O., M. Sarjakoski, and S. Sippu, "System HLP84 — A Tool for Compiler Writing," *Manuscript*, Department of Computer Science, University of

Helsinki (1982).

[Reis84] Reiss, S.P., "An Approach to Incremental Compilation," *ACM SIGPLAN Notices*, **19**(6), 144-156 (1984).

[Reve83] Révész, G.E., *Introduction to Formal Languages*, McGraw-Hill, NY (1983).

[Sebe85] Sebesta, R.W. and M.A. Taylor, "Minimal Perfect Hash Functions," *ACM SIGPLAN Notices*, **20**(12), 47-53 (1985).

[Szaf86] Szafron D. and B. Wilkerson, "Some Effects of Graphical User Interfaces on Programming Environments," *Proceedings of the CIPS/ACI Congress '86*, (1986).

[Szaf88] Szafron D. and B. Wilkerson, "The Smalltalk-80 MVC Paradigm with Plugable Views," *Technical Report TR 88-8*, Department of Computing Science, University of Alberta, Edmonton, AB (1988).

[Teit81] Teitelman, W. and L. Maister, "The Interlisp Programming Environment," *Computer*, **14**(4), 25-34 (1981).

[Trem85] Tremblay, J.P. and P.G. Sorenson, *Theory and Practice of Compiler Writing*, McGraw-Hill, NY (1985).

[Wait85] Waite, W.M., "Treatment of Tab Characters by a Compiler," *Software — Practice & Experience*, **15**(11), 1121-1123 (1985).

[Wait86a] Waite, W.M., "The Cost of Lexical Analysis," *Software — Practice & Experience*, **16**(5), 473-488 (1986).

[Wait86b] Waite, W.M., V.P. Heuring, and R.W. Gray, "GLA — A Generator for Lexical Analyzers," *Software Engineering Group Report No. 86-1-1*, Department

of Electrical and Computer Engineering, University of Colorado, Boulder, CO (1986).

[Wegn87] Wegner, P., "Dimensions of Object-Based Language Design," *OOPSLA '87 Proceedings*, 168-182 (October 1987).